Computer Science

Peter Nyberg
Ulf Larson

Jini Applications in Bluetooth Networks

Bachelor's Project

2001:22

Jini Applications in Bluetooth Networks

Peter Nyberg
Ulf Larson

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Peter Nyberg

Ulf Larson

Approved, 6 June 2001

Advisor: Stefan Alfredsson

Examiner: Stefan Lindskog

Abstract

This report provides an introduction to the Jini, RMI and Bluetooth technologies and describes the basic components and features of each technology. It describes how a Jini application works in a Bluetooth network and states the benefits gained from having this particular combination. It also describes the design, construction and startup procedures of a complete Jini application.

The report is intended for readers wanting to widen their knowledge within some or all of these technologies. Programmers wanting to learn how to design distributed applications by using Jini and RMI may also find it useful.

Acknowledgements

We would like to thank Marcus Andersson at Fyrplus Mekatronik AB and Stefan Alfredsson at Karlstad University, Computer Science department for their supervision in our bachelor project work. We would also like to thank our families Pauline and Therese.

Peter Nyberg
<peter@jazzgame.com>

Ulf Larson
<ulf_larson@home.se>

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Communication is a keyword in todays society. People communicate of social and business reasons and as technology evolves, the number of available communication channels increases. In an extract from Websters Revised Unabridged Dictionary [19] we find the definition of communication to be:

> "Intercourse by words, letters, or messages; interchange of thoughts or opinions, by conference or other means."

Today we use cellphones, mail, fax and chat to communicate. The basic concept is still the same, but technology has largely extended our possibilities of communicating. Technology has also made it possible, not only to easier communicate between people, but also to communicate on a man-machine level. These days we need to communicate with our machines in order to obtain status reports, update information, read news and play games.

Since the wireline system leaves a lot of cables all around and the mobility is limited to the length of the cable, the wireless concept has lately gotten a lot of attention. One relatively new technology that operates in the wireless area is Bluetooth. Bluetooth is designed to connect devices in short distances[1] and to be a cable replacement in networks. Naturally, the wireless concept will make communication easier.

[1] Between 0.1 and 100 m.

Another way of making communication easier is the concept of dynamic service allocation. Imagine that instead of having device drivers installed the traditional way, i.e. floppy disks that mysteriously disappear, download sites that have expired and complex manuals, the services[2] introduce themselves via an interface when there is a need for it. This means that a simple cellphone fairly easy could take the role of a universal remote control and control all enabled devices in its surroundings without having a single driver installed. This technology is also developing at this moment and one of the developers is Sun,[3] working on the Jini technology.

To utilize the existing technology to a maximum would be to construct an application that used both wireless communication and dynamic service allocation. Such an application would have the prerequisites to be very easy to use and at the same time very powerful.

Representative for this kind of application would be one that used both audial and visual features in its appearance. Having a remote control manage the actions of a server would let the user:

- use the remote to *see* what kind of device that is active and also how it can be used.
- *hear* that something happens when the interface is manipulated.

An appropriate application would let the user choose music from a list sent from the server and the server would then play the choosen songs. This kind of application is called a music jukebox.

This report treats the construction of a music jukebox server and a remote control client using the Jini and Bluetooth technologies. The report is divided into eight chapters. The rest of the report contains the following: Chapter 2 describes what conditions had to be met before the project started, it also states the demands on the project. The next three

[2]TV-sets, stereodevices, computers.
[3]Sun microsystems, http://www.sun.com. Jini features are located at http://www.sun.com/jini.

chapters provides the necessary background information for the project. Chapter 3 introduces the Bluetooth concept, explains what a *piconet* is, how the Bluetooth stack is constructed and what types of protocols that are used. Chapter 4, RMI, treats serialization, distributed computing using the RMI model and the *naming* concept. Chapter 5 explains the Jini model, how it is used, what a *service* is, why the lookupservice is important and describes the basic Jini concepts. Chapter 6 describes the use of Jini in a Bluetooth network. Chapter 7 treats the design phase. It describes the Java classes[4] the application consists of. The chapter is illustrated with detailed UML class diagrams of client and serverside. Chapter 8 explains how to run the application and how to interpret the commandlines. The chapter also shows a startup example from the development environment. Finally, conclusion and summary is presented in chapter 9. All figures in the Bluetooth, Jini and RMI chapters are taken from related documentation.[5]

[4]Building blocks in the Java language.

[5]Specifications, whitepapers, webpages and the reference literature.

Chapter 2

Prerequisites and demands

This chapter provides information regarding the prerequisites and demands made on the system and the specific goals stated by the company. It begins with a description of what conditions had to be met *before* the project started, i.e. it states the necessary hardware and software components that had to be present. Then the demands on the expected result of the project are stated together with the type of environment in which the application is supposed to work.

2.1 Prerequisites

In order to initialize the project some important components were required to be present. These components also needed to be operational or at least to be configurable and in working order before the end of the project. Below is the list of prerequisites.

- Operational version of the Java Runtime Environment supporting at least the AWT and RMI classes.
- Operational version of the Jini classes.
- Bluetooth cards providing a wireless connection between two entities.

- Manuals, books and specifications regarding the RMI, Jini and Bluetooth technology.

The Java Runtime Environment that is used is part of the JDK 1.3[1] distribution. This version and the related documentation were present at project initialization time. The Jini classes were downloaded[2] during initialization time. The Bluetooth cards are democards owned by Fyrplus AB. Most of the manuals and books were present at initialization time, additional reading were downloaded.[3]

2.2 Demands

Demands made on the project included a working Jini-application consisting of a client and a server program. The list of demands are stated below.

- Writing brief overviews of the RMI, Jini and Bluetooth technologies in order to increase knowledge of these subjects.
- Designing and implementing an operational Jini-application consisting of client and server programs.
- Evaluating the possibilities of configuring the link using the provided Bluetooth hardware.
- Configuring a Bluetooth link to support the wireless communication between client and server (assuming this is possible). The configuration and usage of the Bluetooth link depends on the result of the third item, if it is not possible to configure the Bluetooth link then a WLAN link will be used for the connection.
- Demonstrating the working wireless Jini-application.

[1]The Java Developer Kit version 1.3 is a free software package that is downloadable from www.java.sun.com/j2se.

[2]From www.sun.com/jini.

[3]See reference section for addresses.

2.3 Purpose

The goals of the project that Fyrplus AB intended to reach. Below are the goals stated:

A framework for Jini applications: To obtain a generic infrastructure for further development of Jini applications.

Documentation of the RMI, Jini and Bluetooth technologies: To learn more about RMI, Jini and Bluetooth in order to develop future applications by using these technologies. Also, by having brief introductions and examples there is no need to read the full specifications to understand the technologies. I.e. basic development can begin with relatively little reading.

A working application: Having a working application in order to demonstrate the functionality of ad hoc applications.

Extended knowledge of the "ad hoc" concept: Obtaining proof of the advantages of having ad hoc networks and services.

Evaluation of the Bluetooth hardware: Testing the Bluetooth hardware owned by Fyrplus AB and evaluating the possibilities of configuring a Bluetooth link.[4]

[4]See Section 6.2 for further reading on the result of the link configuration.

Chapter 3

Bluetooth

This chapter provides a brief introduction to the Bluetooth technology and is divided into five sections.

The first section, "What is Bluetooth?", is a short introduction to the technology. Section two, "Usage models" states some of the different usage models for Bluetooth, for example; the cordless computer and the interactive conference.

Section three, "Piconets, the master-slave role", describe what a piconet is used for and why it is an important concept. This section also describes the scatternet concept.

Section four, "The technology in brief", describes how the message passing procedure works, it also explains what *frequency hopping spread spectrum* is.

The last section, "The protocols", is an indepth explanation of the Bluetooth standard, it describes the protocols defining the Bluetooth stack.

3.1 What is Bluetooth?

The Bluetooth standard is a new technology using short-range radiolinks intended to replace the cable connection(s) between nodes in a network. These links build an ad hoc

network[1], called piconets. Piconets can constitute of up to eight units and each unit can communicate with other piconets as well. The piconets are established dynamically and automatically as Bluetooth devices enter and leave the radio proximity.

The Bluetooth idea first emerged at Ericsson who formed a Special Interest Group (SIG) for Bluetooth with the task of making a standard for wireless radio communication with as many of the big telecommunication companies as possible involved. The specification is free and open to all, making it easier for developers and programmers to learn and use the technology an thus making it more available and probably make the market for Bluetooth bigger. The logotype of Bluetooth is shown in Figure 3.1.1.

Key features that makes Bluetooth unique is its ability to operate on low power, relatively cheap devices, low complexity and robustness. Another key design issue was reuse of existing higher layer transport and application protocols already developed. The radio links replace the cables in networks on the lower levels[2] in the protocol stack and makes the services unaware of the absence of cables. The network is well suited for mobile devices which communicate over the ether, with minimal user effort. The technology offers wireless access to LANs, the mobile phone network, the Internet and handheld devices. Bluetooth uses *spread spectrum*[3] for usage of the frequency band which is based upon frequency, time and coding schemes.

Essentially Bluetooth is a standard describing a short-range frequency hopping radio link between devices. Bluetooth is split into two parts, the Bluetooth Specification and the Bluetooth Profiles.

[1]See Chapter 6 for further reading.
[2]The physical layers: Bluetooth radiolayer and Bluetooth baseband.
[3]See Section 3.4 for further reading.

Figure 3.1.1: The Bluetooth logotype.

3.2 Usage models

Some of the different usage models are formally specified and described in the *profiles specification*[4]

The Cordless Computer: This is the main feature of Bluetooth, the cable replacement, the foremost reason for developing the standard. Wireless networking can offer more freedom in placement and use of computer devices.

The Headset: The Bluetooth headset could be used as handsfree for the mobile phone or the stationary phone. The headset could connect through the stationary phone. This makes the user free to roam the area while still keeping the connection intact. The headset can also be used for multiple devices. This makes the placement of devices easy since devices can be placed wherever convenient.

The three-in-one phone: This feature enables the usage of the phone as cellular phone, as a cordless home phone connected to the voice access point and as a walkie-talkie for direct phone-to-phone communication when the devices are within proximity. The maximum range of the walkie-talkie is 100 meters, see Table 3.5.2 for further information on the ranges of Bluetooth devices.

The interactive conference: File transfer is one of the key features of computer networking. The interactive conference could use wireless network to exchange business

[4]The profiles are not further discussed in this document but are fairly similar to the usage models. See [6] for further reading.

cards and files to the participants of the meeting. The Bluetooth link could also be used for data exchange between two devices without the configuration of the network.

The Internet bridge: The bridge could be a link between the portable computer and a LAN using a data access point. The access point is a "wireless plug" to the network. The link could also be a wireless data modem using a telephone. The phone connection is essentially the same thing as dialing on to the web with your dial-up modem without cables.

The automatic synchronizer: The PDA and the cellular phone might use the Bluetooth link to automatically synchronize with all handheld devices as well as the stationary computer and its calender, to-do list, address book and so on when the devices come within proximity. This makes the change in one of the devices "transfer" the change to all the other Bluetooth units.

The instant postcard: The idea is having a digital camera take a picture and transfer the picture as an e-mail to some recipient as a digital "postcard". This is done through wireless communication between the camera and some Internet connected access point.

Ad hoc networking: Spontaneous networks between all kind of Bluetooth devices within proximity.

Hidden computing: The automatic synchronization is one example of this idea. A notebook computer "hidden" in a briefcase could peroidically receive and/or transfer e-mail to a mobile phone. Or the other way around, a notebook computer wants to connect to the internet and uses the mobile phone who is "hiding" in the briefcase.

3.3 Piconets, the master-slave role

All Bluetooth network consist of a number of piconets talking to each other. Two or more units that communicate form a piconet. One unit act as master of the piconet and the other act as slaves. There can be only one master in each piconet but this master can act as a slave in another piconet. A piconet can only include up to seven *active* slaves and many more *parked* slaves. Piconets that overlap form scatternets.

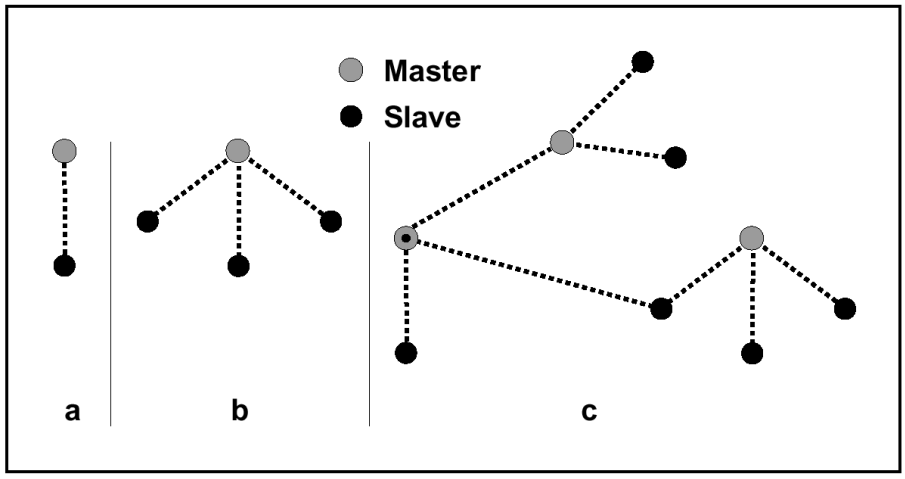

Figure 3.3.1: Piconets with a single slave(a), a multi-slave operation (b) and a scatternet operation (c).

There are four modes for connected piconetworks (active, sniff, hold and parked) as well as one standby mode.

- In *active* mode the slave listens for transmissions from the master.
- In *sniff* mode a slave becomes active periodically.
- In *hold* mode the slave may stop listening for packets entirely for a period of time.
- In *parked* mode the slave is not considered to be active any more.

The modes are further explained in Section 3.5.2 (Connection modes).

The master owns the master clock which all the units in each piconet use for the *frequency hopping spread spectrum* (FHSS)[5] synchronizated communication between the units. The master role does not imply any special privileges or authority. Generally the unit who initiates the session takes the master role. The hop sequence is unique for each piconet. All units in a piconet is time- and hop-synchronized.

3.4 The technology in brief

The use of division spread spectrum splits messages into packets and sends the first packet in one channel, the next packet is transmitted on a new channel. This spreads the message over the available frequency band. The receiver on the other hand must know the hopping pattern to be able to tune into the right channels and recieve each packet and reassemble them into the original message. This process is called *frequency hopping spread spectrum* FHSS.
The benefits of FHSS is:

- The interference caused by colliding transmission on the same frequency is less likely than if each radio used one single channel for a long duration.
- If however a collision does occur, the effect is minimal, since only a single packet is lost and can easily be retransmitted at a new frequency.
- FHSS can provide some degree of security in the communication since only the receiver who knows the hopping pattern can reassemble all the packets of a message. Thus FHSS can be employed to hinder eavesdropping.

The Bluetooth protocol uses a combination of circuit and packet switching. Bluetooth operates in the Industrial, Scientific and Medical (ISM) band at 2.4 GHz. A slotted channel with nominal slot lenght of 625μs is used. For full-duplex communication the Time-Division

[5]FHSS is further explained in the next section.

Duplex (TDD)[7] scheme is used. Packets are transmitted on different hop frequencies and these packets nominally cover one time slot but can cover up to five slots. The Bluetooth standard supports point-to-point and point-to-multipoint connection. Slots can be reserved for the synchronous packets, such as audio packets, which garanties a certain bandwidth. A minimum bandwidth for the asynchronous packets can be negotiated through the Link Manager Protocol (LMP), described in Section 3.5.3.

3.5 The protocols

Further reading is a more indepth explanation of the standard and is only urged for people specially interested in the technology.

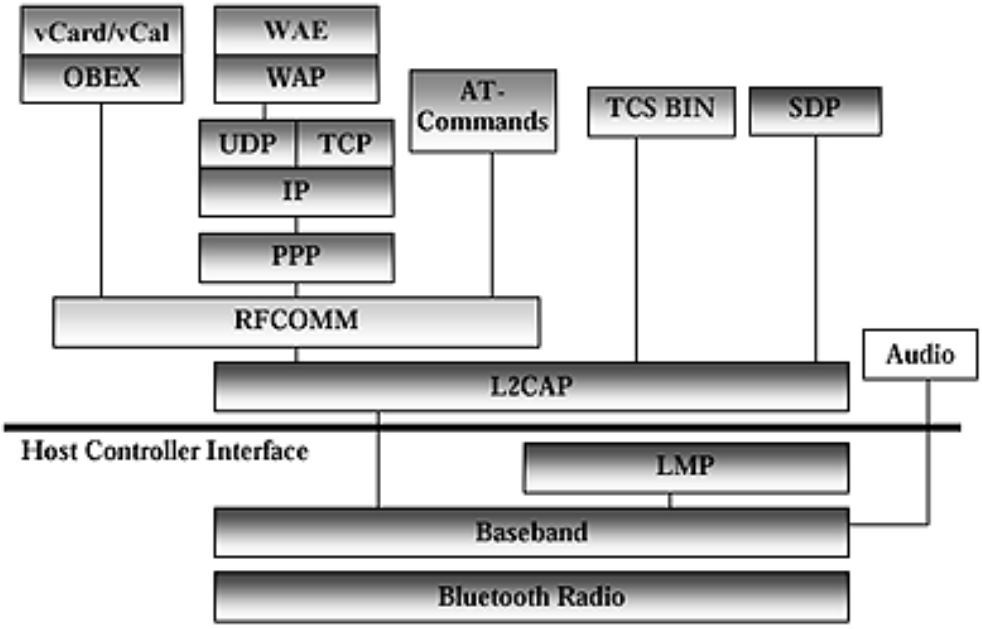

Figure 3.5.1: The Bluetooth stack.

The protocols[8] have been segmented into almost independent layers that make a good abstraction of the functionality of the Bluetooth model. The main principle in creating

2.4 GHz ISM band (GHz)	frequency channels (GHz) k=0,1,...,m-1	Low guard band (GHz)	High guard band (GHz)
2.400-2.4835	2.402+k ; m=79	2.0	3.5

Table 3.5.1: The ISM frequency band.

the protocol stack has been to maximize the re-use of existing protocols for the higher levels. Different applications may run over different protocol stacks, nevertheless each one uses a common Bluetooth physical link (i.e. the Bluetooth radiolayer and the Bluetooth baseband layer).

3.5.1 Bluetooth radio layer

The radio layer form the physical interface to other Bluetooth units. There are some problems/restrictions on the ISM radio band:

- channel bandwidth is limited to 1 MHz
- multiple networks may exist and interfere
- microwave ovens use this band

Bluetooth solves the design obstacles using spectrum spreading by frequency hopping (FH) with 79 hops displayed by 1 MHz around the ISM frequency 2.4 GHz. The nominal hop rate is 1600 hops per second. Bluetooth utilizes the maximum number of channels availible, 79 in most countries and 23 in the rest (France at the moment). With this hopping sequence the time slots is 625 μs per hop. See Table 3.5.1 for a more further information on the ISM frequency band.

Modulation in the Bluetooth radio module uses Gaussian Frequency Shift Keying (GFSK)[14]. In GFSK a binary one is represented by a positive frequency deviation and a zero is represented by a negative frequency deviation. The error rate may be high, especially due

to strong interference from microwave ovens which operate at this frequency, therefore Continous Variable Slope Delta Modulation (CVSD)[7] coding has been adopted for voice, which can withstand high bit error rates. The packet headers are protected by a highly redundant error correction scheme to make them robust to errors.
The radio specification defines three power classes:

Power classes	*Range*	*Output power*
Power class 1	long range (100m) devices	20 dBm
Power class 2	ordinary range (10m) devices	4 dBm
Power class 3	short range (10cm) devices	0 dBm

Table 3.5.2: The power classes of the Bluetooth technology.

Each device can optionally vary its transmit power.

3.5.2 Bluetooth baseband

This is the physical layer in the network stack. The baseband protocol[7] is implemented as a link controller, which works like the link manager carrying out link level routines such as link connection and power control. The baseband provides the functionalities required for devices to synchronize their clocks and establish connections between the Bluetooth units. Inquiry procedures for discovering the addresses of devices in proximity are also provided. The packet handling over the wireless link is the responsibility of Baseband. Baseband also takes care of lower level encryption for secure links. All packets can be provided with different kinds of error correction and encryption. Audio is relatively simple to send to and from a Bluetooth unit by just opening an audiolink. The frequency hop sequences are provided by this layer.

Physical channel: The 2.4 GHz ISM band is used for transmission. 79 channels divided on 1 MHz of bandwidth are used in most countries. The frequency hopping of the channel is set up by the master of the piconet and its BD_ADDR[6] and clock. The

[6]The Bluetooth Device Address.

physical channel is divided into time slots where each slot represents a hop frequency. The time slots are 625 μs in length, which makes about 1600 slots per second. The TDD scheme is used for alternative transmission of the master and slaves. The master starts its transmission in even-numbered slots only, the slaves can only start its transmission in odd-numbered slots. The packet starts must be aligned with the time slot start.

Physical link: There are two types of physical links defined. The SCO (Syncronous Connection-Oriented) link and the ACL (Asyncronous Connection-Less) link. ACL packets are used for data only, while the SCO can contain audio only or a combination of audio and data. The SCO link is point-to-point between master and one slave in a piconet. The master administers the SCO link in reserved slots at regular intervals (circuit switched type). The SCO packets are mostly used for audio and does not use any more protocol higher than baseband. ACL links are point-to-multipoint between master and single/multiple slaves of a piconet. Slots not reserved for SCO packets are used for ACL packets. Even slaves that already have a SCO link can establish a new ACL link to the master. Retransmission is mostly used for ACL packets.

Logical channels: There are five different kinds of channels used for different types of information:

- LC (Logical Control channel) *contains low link level info. It is mapped onto the packet header.*
- LM (Link Manager channel) *carries control info exchanged between link managers of the master and slave(s), can be carried by SCO or ACL in the payload.*
- UA (User Asynchronous data channel) *carries Logical Link Control and Adaption Protocol (L2CAP)*[7] *transparent asynchronous user data. Normally an ACL link.*

[7]See Section 3.5.4 for further reading.

- UI (User Isosynchronous[8] data channel) *carries L2CAP transparent isosynchronous user data. Normally an ACL link.*
- US (User Synchronous data channel) *carries L2CAP transparent synchronous user data. Is carried in SCO links only.*

The logical channels are sent as payload, except LC.

Device addressing: There are four different kinds of addresses that a Bluetooth device can be assigned:

- BD_ADDR (Bluetooth Device Address) each tranceiver is allocated a unique 48-bit address. It is divided into a 24-bit Lower Address Portion (LAP) field, a 16-bit Non-significant Address Portion (NAP) field and a 8-bit Upper Address Portion (UAP) field. This address is electrically "engraved" on each device.
- AM_ADDR (Active Member Address) a 3-bit number which is only active as long as the slave is active in the piconet. This address is sometimes called the Medium Access Control (MAC) address of a Bluetooth unit.
- PM_ADDR (Parked Member Address) a 8-bit member address that separates the parked slaves (i.e. there can be 256 parked slaves). This address is only valid while the slave is parked.
- AR_ADDR (Access Request Address) this address is used by the slave to determine the slave-to-master time slot, access window and where the slave is allowed to send access request messages to the master. This address is not necessarily unique.

BT clock: Each Bluetooth device has a 28-bit native clock running. This clock is never adjusted or turned off. The clock ticks every 312.5 μs this represents a clock rate of 3.2 KHz. The slaves of piconets uses the master clock for communication and

[8]Synchronized packet with a time dependency.

synchronization. The BD_ADDR and the master clock is used to get the offset for each device.

Packet format: The packets physical appearance and the meaning of the fields can be seen in Figure 3.5.2.

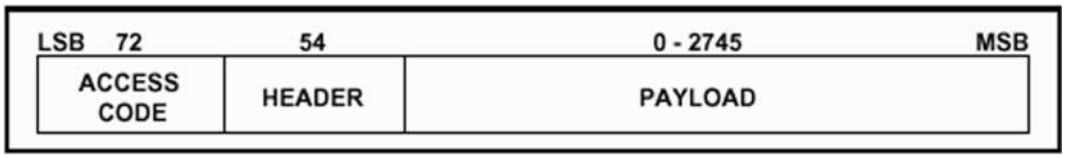

Figure 3.5.2: A Baseband packet.

- **Access Code:** used for timing synchronization, offset compensation, paging and inquiry. There are tree different types: Channel Access Code (CAC), Device Access Code (DAC) and Inquiry Access Code (IAC). CAC identifies a unique piconet while the DAC is used for paging and its responses.
- **Header:** contains information for packet acks, numbering of packets, flow control info, slave address and error check for header.
- **Payload:** can contain data or voice information or both. It will also contain a payload header.

Packet types: All data of the piconet is conveyed in packets. There are 13 different packet types. All higher levels of the stack use these packet types to compose messages. The packets are: ID, NULL, POLL, FHS, which are defined for both SCO and ACL packets, DH1, AUX1, DM3, DM5, DH5 are defined for ACL packets only and finally HV1, HV2, HV3, DV are defined for SCO packets only.
A short description of the packet types:

- ID: 68-bit packet for paging, inquiry and response routines. Essentially the DAC or the IAC.

- NULL: 126-bit packet containing the CAC and header. Used for returning link information to the source.
- POLL: Like the NULL packet but it needs an ack from the destination. Upon reception of a POLL packet the slave must respond with a packet.
- FHS: 144-bit packet that reveals the BD_ADDR and the clock and the source device. Has a 16-bit CRC code. Payload has a 2/3 rate FEC which brings the payload to 240 bits. The packets cover one time slot.
- DH: Data-High Rate for ACL-links. The numbers 1, 3 and 5 represent the number of slots the packet covers.
- DM: Data-Medium Rate for ACL-links. Carries data inforamtion only. A 16-bit FEC and use a 2/3 FEC encoding. The numbers 1, 3 and 5 represent the number of slots the packet covers.
- HV: High quality Voice. A SCO-link packet. Uses a 1/3 FEC for HV1, 2/3 FEC for HV2 and HV3 use no FEC. HV packets does not use CRC or payload header.
- DV: Data Voice. A SCO-link data and voice packet. 80-bit voice field and 150-bit data field. The voice has no FEC, but the data field has a 2/3 FEC. The two fields are treated differenetly as the voice is never retransmitted, it is always new. The data field is checked for error and might be retransmitted if necessary.
- AUX1: An ACL-link packet for data. Ressembles a DH1 packet but has no CRC code.

All operations in a Bluetooth network are related to two fundamental elements: the Bluetooth device address and Bluetooth device clock.

Channel control: A connection between two devices can occur either by both the inquiry

and page procedure or if some details are known about a remote host then only the paging procedure is needed. Paging is the process of inviting a device to join the piconet. The inquiry procedure enables a device to discover which devices are in the proximity and determine the address and clock of these devices.
Bluetooth operates essentially in two different states: *Standby* and *Connection*. Standby is the default state of any Bluetooth device. In this state only the clock is running and there is no communication with other devices. In the Connection state master and slaves can communicate over the Bluetooth link. There are seven substates that Bluetooth use to administer the piconets. These are:

- **page**, the master explicitly invites a new device to the piconet. The device must be in page scan mode to listen to and subsequently respond to a paging.
- **page scan**, the device is ready, listening for paging with its own DAC.
- **inquiry**, the device learns about its neighbouring devices. These devices must be in inquiry scan mode to listen to and subsequently respond to an inquiry.
- **inquiry scan**, the device is ready, listening for inquiries.
- **master response**, first the master recieves a reply to the page message and sends a FHS packet to the destination. Second the master has recieved a reply to the FHS packet and knows that the identities in the piconet is set.
- **slave response**, the slave has received its DAC from the master and sends a response. When the slave has received the FHS from the master it sends a reply the DAC again in a ID packet. Finally the slave switches to the masters channel.
- **inquiry response**, this is when the device has received inquiry packets and responds with a inquiry response.

The Connection state starts with a POLL packet from the master to verify that the slave has switched to the right clock offset and channel frequency.

Connection modes: There are four baseband modes for connected networks (active, sniff, hold and parked) as well as one standby mode.

- In active mode the slave essentially always listens for transmissions from the master.
- In sniff mode a slave essentially just becomes active periodically. This is a method of reducing power consumption. The master and slave agree to transmit for a period of time. The slave only listens for packets in the beginning of each interval.
- In hold mode the slave may stop listening to packets entirely for a while. The master and slave agree on this time for which the communication link between these two are down.
- In parked mode the slave maintains synchronized with the master, but is not considered to be active any more.
- In standby mode the device is not a member of a piconet.

Flow control: The baseband protocol recommends the use of FIFO queues in ACL and SCO links. The Link Manager fills the queues and the link controller empties the queues automatically. If the FIFO queue is full the link contoller of the receiver inserts a *stop* indicator in the header of the return packet. The transmitter freezes its FIFO queue when a *stop* is received. When the receiver is ready it sends a *go* packet which resumes the normal flow again.

3.5.3 LMP - Link Manager Protocol

The link manager in communicating devices exchange messages to control the Bluetooth link between these devices. It carries out setup, authentication and link configuration. The communication with other link managers is realized via the LMP[12]. The LMP essentially

consists of a number of Protocol Data Units (PDU), which are sent between devices. It should be noted that PDUs are not executed in real-time like baseband operations. Device authentication is a mandatory feature supported by all Bluetooth devices. Link encryption is optional. Authentication depends on a shared secret. The encryption of a Bluetooth wireless link is based on a 1-bit cipher stream, whose implementation is included in the specification. One of the features of the link setup is power management, occationally a device might request a partner to adjust its transmission power depending on the quality of the link, as measured by the strength of the incoming transmissions. There are three low power link modes: *sniff, hold* and *parked* (explained in the previous section of this chapter), all of these modes are optional. There is some form of quality of service that the LMP can negotiate, for example; the minimum bandwidth assignment for ACL links between two devices by setting the maximum polling interval for the link.

3.5.4 L2CAP - Logical Link Control and Adaption Protocol.

The L2CAP[13] offers peer-to-peer connection both connection-oriented and connectionless data services to upper protocol layers. This layer shields the higher-layer protocols from the details of the physical link. Thus the higher levels need not know of the frequency hopping nor of the packet formats in the baseband and radiolayers. The L2CAP protocol supports protocol multiplexing, allowing multiple protocols and applications to use the same Bluetooth air-link. It also takes care of assembly/reassembly of higher level packets into/from the smaller baseband-packets. The L2CAP permits transmit and receive data packets up to 64 kB in length. The baseband supports both SCO[9] and ACL links but L2CAP is only specified for ACL links. If no Host Control Interface (HCI)[10] is present the applications would interact with this protocol.
The basic L2CAP functions:

[9]The SCO and ACL packets are further explained in Section 3.5.2.
[10]Further explained in Section 3.5.5.

Multiplexing: The protocol must allow multiple applications to one link between two devices simultaneously.

Segmentation and Reassembly: The protocol must resize the packets from the higher layers to the size accepted by the Bluetooth baseband. The reassembling of the segmented packets is also the protocols responsebility. L2CAP accepts packets of 64 kB but Baseband only supports 2745 bits (343 bytes).

Quality of Service: Certain parameters like peak bandwidth, latency and delay variation can be negotiated. L2CAP checks if the link is capable of the requested service and provides the service if possible.

Groups: The concept of grouping of addresses. This concept in Baseband is supported through the piconets. L2CAP needs to make an abstraction of the piconets to the higher protocols. The implementations of the higher protocols can map on to piconets.

Figure 3.5.3 illustrates the events and actions performed by an implementation of the L2CAP layer.

L2CAP is a packet based layer and follows a communication model based on channels. A channel represents a data flow between two L2CAP entities on the peers. Channels can be connectionless or connection-oriented. All the packet fields over the communication channel use Little Endian[11] byte order. L2CAP does not enforce a reliable channel, i.e. no retransmissions or checksum calculations are performed. Multiple channels can be bound to the same protocol, but a channel can not be bound to multiple protocols. This means that the protocol must allow multiple applications (which might use different protocols) to use a link between two devices.

[11]Bytes are sent from right to left, the "little" LSB bit is transmitted first, for example the Pentium is little endian. See [23] for further reading regarding little endian.

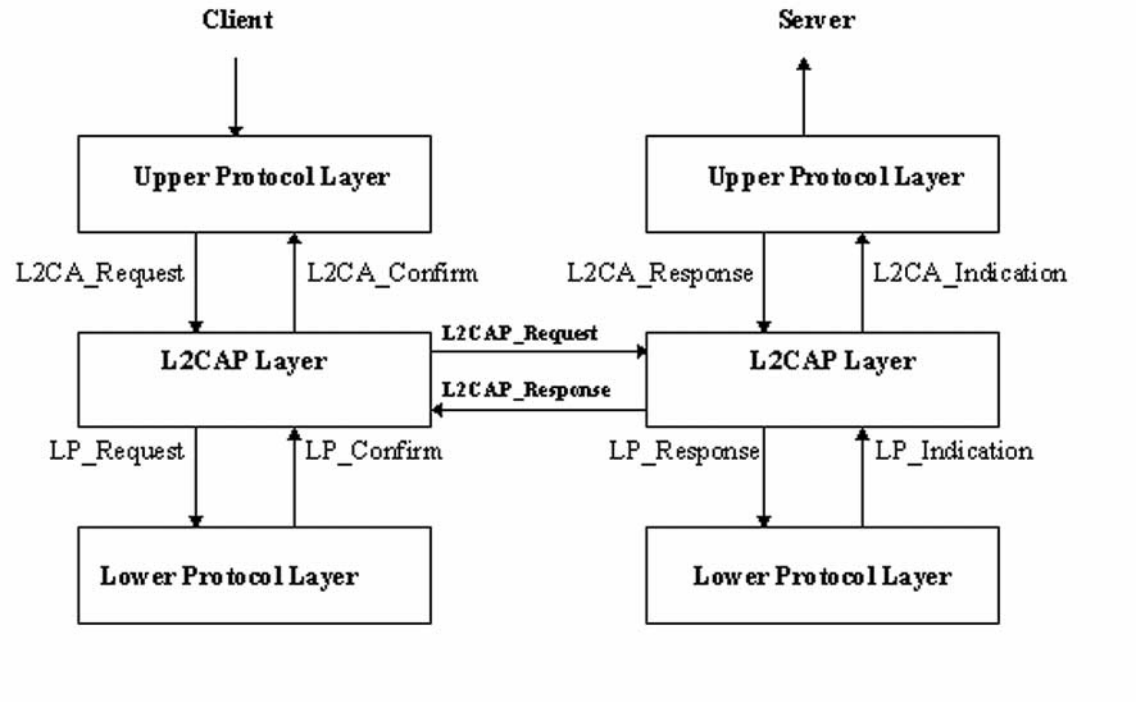

Figure 3.5.3: The state machine used in L2CAP protocol layer.

There are some services offered by L2CAP in terms of service primitives and parameters. These are listed below:

- **Connection**: setup, configure, disconnect
- **Data**: read, write
- **Group**: create, close, add member, remove member, get membership
- **Information**: ping, get info
- **Connection-less traffic**: enable, disable

3.5.5 Host Control Interface

The HCI[10] allows higher levels of the stack (L2CAP and above, see Figure 3.5.1), including applications, to access the baseband, link manager and other hardware registers

through a single interface. To permit L2CAP and higher protocol layers and applications to transfer and receive control and application data from any Bluetooth unit the SIG has developed a protocol to access the hosts in a uniform manner, the HCI protocol. The HCI protocol exposes the internal operations of the lower transport protocols.

In many devices, the Bluetooth enabling module may be added as a separate card. For example as a PCI card or a USB adapter. Hardware modules usually implement the lower layer radio, baseband and LMP. The module attaches to some kind of host, enabling that device with Bluetooth wireless communication. The data travels over a physical bus like USB. A driver for this bus is required on the "host" and a HCI is required on the hardware card to accept data over the physical bus. Thus, if the higher Bluetooth layers (L2CAP and above) are in software and the lower ones in hardware, these layers are at least required:

- **HCI driver**: the driver, residing in the host, the software entity, above the physical bus, formats the data to be accepted by the HC on the hardware. Host means a HCI-enabled software unit.

- **HCI**: resides in the hardware and accepts communication over the physical bus.

HCI uses three different HC transport layers. These three transport layers should provide the ability to transfer data transparently. The host should receive asynchronous notifications of HCI events independent of which transport layer is used. The layers are:

- **RS232**: the objective is to make it possible to use the Bluetooth HCI over one physical RS232 interface between the Bluetooth Host and the Bluetooth Host Controller. Event and data packets flow through this layer, but the layer does not decode them.

- **USB**: the objective is to use a USB hardware interface for Bluetooth hardware. A class code will be used that is specific to all USB Bluetooth devices (a generic driver). This will allow the proper driver stack to load, regardless of which vendor built the

device. It also allows HCI commands to be differentiated from USB commands across the control endpoint.

- **UART**: the objective is to make it possible to use Bluetooth HCI over a serial interface between two UARTs on the same machine, i.e. connection to a serial port without the need of a serial cable. The UART is a proper subset of the RS232 protcol. Event and data packets flow through this layer, but the layer does not decode them.

3.5.6 RFCOMM

The RFCOMM[15] is an emulation of a serialport. Serial communication is an important feature in the cable replacement idea. RFCOMM presents a virtual serial port to applications. The migration to wireless communication for applications is now a simple task since the use of the serialport is commonly known. RFCOMM resides directly on top of the L2CAP layer see Figure 3.5.1 for illustration. Requirements for Bluetooth serial communication:

Multiplexed serial communication: Many simultaneous clients of the serial interface in the stack.

RS232 compatibility: RS232 is a widely used serial interface for wired networks, which Bluetooth aims to replace. Many applications are used to the RS232 control signals such as: Request/Clear to Send (RTS/CTS), Data Terminal/Set Ready (DTR/DSR) and the RS232 break signals. Emulation of these signals allows RFCOMM to supply the clients with a serial port that is virtually the same as that of the cable.

Remote status and configuration: The configuration of the peer-to-peer connection needs to be negotiated so the peers are compatible. The Service Discovery Protocol (SDP), discussed in the next section, can be used for some basic information the serial channel needs.

Internal and external serial ports: To support serial communication the RFCOMM needs the emulation of an internal serial port, i.e. only locally, as well as an external serial port where parameters and status are transmitted across the RF link.

RFCOMM uses a L2CAP connection to instatiate a logical serial link between two devices. Only a single RFCOMM connection is allowed between two devices at a given time, but the channel may be multiplexed so that there can be multiple logical serial links between the devices. Each multiplexed link is identified by a number called the *Data Link Connection Identifier*, or DLCI. The specification allows up to 60 multiplexed links. Consider the network access point that allows multiple Bluetooth devices to access a larger network[12] this kind of usage model might have many clients connected to the access point. It is important that each logical channel with its own set of data and control signals be multiplexed to a single channel so that no mixup between logical channels are made.

3.5.7 SDP - Service Discovery Protocol

The classical static configuration of printer, mouse, etc is not a good idea for ad hoc networking like Bluetooth. The Bluetooth community need a more dynamic way of locating the available services in a network, this is called service discovery. The SDP[16] addresses a way of telling the network what services each unit owns and also to learn what services other units own, dynamic location of services. This is a key feature in making value of the network to the end-user. Since service discovery is fundamental to all Bluetooth profiles, most applications will use the SDP layer, see Figure 3.5.1 for reference. This layer helps the devices to *self-configure*, i.e. discover each other and negotiate what they need to do and which devices needs to collaborate and this is done without any human intervention. Some key design features of the SDP:

Simplicity: Since service discovery is a part of nearly every Bluetooth usage case, it is desirable that the discovery process is as simple as possible to execute.

[12]This issue is dealt with in the LAN Access profile of the Bluetooth specification.

Compactness: Service discovery is a typical operation performed soon after a link is established. The SDP air-interface traffic should be minimal so that service discovery does not unnecessarily prolong the initialisation process.

Flexibility: There are many usage models that use and will come to use the Bluetooth technology. Since not all profiles and usage models can be foreseen, it is important for SDP to easily be extended to accommodate the new services that will deploy in the Bluetooth enviroment.

Service Class and Attributes: In a dynamic network it is important for clients to quickly locate a specific service when they know what they want. The scenario of looking for a "printer" class in the network should be straightforward as well as a "printer" with the "Epson" attribute or some kind of physical location attribute.

Service browsing: It can also be of interest for the client to know what services are available in the Bluetooth environment at the moment. This is a different process than searching for a specific service it is looking in the "general"[13] class.

SDP includes the notation of a client (entity looking for services) and a server (registry entity, service provider). Any device might take the role as server or client, compare with the master-slave role in piconets. The registry is a list of service records. A service record is a description of the service in a standard fashion as described in the specification. Discovering a service in Bluetooth wireless communication reduces to a simple operation: the client asks the server for the specific services it is interested in and the server responds with the available services that match the question. One of the design goals of SDP was to ensure that other popular discovery protocols, such as Salutation[25] or JINI, could be used in conjunction with it. One of the things that can be discovered using SDP is that the service supports more discovery protocols. Thus SDP might be used to locate

[13]I.e. looking for all available services and keeping some kind of track of the immediate environment, this can be compared with the lookup-service in the Jini technology.

a service; further SDP transaction might be used to discover that the device supports JINI[14] for example; once this has been discovered the new protocol (JINI) can be used for further interaction with the service. The SIG is working towards formalizing these discovery protocols into profiles.

3.5.8 IrOBEX

OBEX[11] as it is briefly called, is an adopted session protocol developed by the IrDA to exchange data in a simple and spontaneous manner. The purpose of including IrDA[11] protocols in the stack is to promote the interoperatability with IrDA applications. The interoperability is at the application layer. The Bluetooth devices can not directly communcate with other IrDA devices, instead it promotes the devolopment of applications that can use either Bluetooth radio or IrDA infrared transport. Wireless communication can now be supported over both Bluetooth and IrDA links. All OBEX transactions must use a separate RFCOMM channel and thus the protocol multiplexing in the RFCOMM is an important feature. At the moment only support for the RFCOMM layer as transport protocol is supported, but in the future TCP/IP is likely[15] to be implemented as a transport.

3.5.9 TCS and AT-commands

The Telephony Control Specification (TCS) is designed to support telephony functions, including call control and group management. TCS is a binary (sometimes called TCS-BIN) encoding for packet-based telephony control and resides above the L2CAP layer in the stack, see Figure 3.5.1. In voice calls these functions set up the connection and then a Bluetooth audio channel is used for the contents of the voice call. TCS can also be used

[14]The JINI technology is explained more in depth in the Jini chapter of the report.

[15]OBEX operation over TCP/IP links is already enabled in the OBEX standard. Because TCP/IP is such an important protocol it is likely that TCP/IP over Bluetooth (wihtout PPP) will probably soon be solved.

to set up a data channel which later uses L2CAP to transfer the actual data packets. the key features of TCS are:

Call control: Serves to set up calls that subsequently will carry voice or data traffic. Multiple instances of TCS can be executed at the same time to handle multiple calls (recall that the L2CAP can have up to three voice channels simultaneously).

Group management: This feature is useful in telephony applications to be able set up the rules for what functions the application shall have, such as call forwarding and group calls.

Connectionless TCS: This makes it possible for devices to negotiate signaling info without actually making a call. This information call can be audio control: speaker/microphone gain or company information, which is the way devices can exchange non standardized info.

A second kind of call control is AT-commands[16] as seen in Figure 3.5.1. This is not a named protocol in the Bluetooth standard but is a standard way for accomplishing call control and is further used by many profiles. AT applications are configured to communicate with a modem over a serial port. This communication is accomplished through the RFCOMM layer with the same AT commands as in other environments.

3.5.10 Audio

Audio[9] is treated uniquely in Bluetooth. This is because audio traffic is *isochronous* meaning it has a time element associated with it. The importance of time in audio transmission is much greater than that of correctness and traffic is routed directly from the baseband layer, as can be seen in Figure 3.5.1. Special audio packets (SCO-packets) are used for the audio traffic. There can be up to three audio channels at once, with some

[16] AT commands are modem control commands.

bandwidth left for data traffic. Audio traffic takes place at a rate of 64 kbps using one of two data encoding schemes: the 8-bit logarithmic *Pulse Code Modulation* PCM[23] or *Continuous Variable Slope Delta* CVSD[7].

3.5.11 WAP over PPP

In Bluetooth point-to-point is accomplished through the use of the adopted PPP protocol. PPP runs over the virtual serial port (RFCOMM) and makes the abstraction of a regular network connection to the WAP protocol. Access to the TCP/IP[21] stack is operating system independant and is mostly realized through the socket programming interface model. TCP/IP/PPP is used for all the Internet bridge scenarios and for OBEX in the future. UDP/IP/PPP is available as transport for WAP.

3.6 Conclusion of the Bluetooth technology

This section concludes the Bluetooth introduction with a short listing of the most important features previously explained.

- Bluetooth is a wireless technology for cable replacement.
- Bluetooth SIG is a forum for developing the technology, main participants of the SIG are for example Ericsson, Nokia, Intel and IBM.
- The Bluetooth chip is a small, cheap radio chip to be plugged into computers, printers, mobile phones, etc.
- Bluetooth RF is in the ISM frequency band (2,4GHz) and uses FHSS.
- There are a lot of interesting usage models; *the headset, the cordless computer, tree-in-one phone, the instant postcard, etc.*

- The protocols are in ascending order (from the radiolayer and upward): *Radio layer, Baseband layer, LMP, L2CAP, RFCOMM.*
- The HCI can communicate with the lower layer using primitives.
- There are many higher protocols defined: *OBEX, SDP, TCS, WAP, etc.*
- Audio can be sent directly over a Bluetooth link without any higher protocols.

Chapter 4

RMI

This chapter provides a brief introduction to the RMI concept; including what RMI is, how it works and for what reasons it was created. It also contains an overview of *object serialization*.

This chapter is divided into four sections: *What is RMI*, *Object serialization*, *System goals and advantages* and *the RMI model*. The "What is RMI" section describes the RMI process at the highest abstraction level excluding almost all details. This section also attempts to provide a clear definition of the RMI concept. "Object serialization" describes what serialization means and why it is used. Section three, "System goals and advantages" states the primary design goals (the reason) of RMI. It also describes some of the advantages emerging from the design. The last section, "The RMI model", describes the RMI distributed application model, the layered architecture and the *Naming* concept. In addition, the difference between definition and implementation (which turn out to be one of the cornerstones in RMI) is explained.

4.1 What is RMI?

RMI is the short form of *Remote Method Invocation*. As the name suggests, RMI is concerned with doing method invocation remote. So, how should this be interpreted? Two statements will help explaining the concept: First, invoking a method means that a Java object makes a call to an arbitrary written Java method belonging to a Java .class file.[1] Second, *remote*, in this context means "located in a different Java Virtual Machine (JVM)." With the aid of these statements a definition of RMI can be stated as follows:

"Remote Method Invocation provides a model for distributed computation using Java objects"[5].

There is no physical limit on the distance between communicating JVM's. They may be located just a few memory cells away as well as on the other side of the planet. The only requirement for successful communication is the presence of a TCP/IP network, and a JVM version supporting the RMI classes.

The highest abstraction level of the RMI concept is the one treating two entities (known as the *client* and the *server*) communicating with the aid of RMI. In this scenario the client entity invokes a method on a server entity object and receives the answer, leaving the delivery details to RMI.

4.2 Object serialization

Java input and output is based on the use of streams. A stream is a sequence of bytes that travel from a source to a destination over a communication path. The path depends on the kind of input/output (I/O) being performed and can be a file system, a network or

[1]For details about the Java language, please refer to the Java Language Specification at http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

another form of I/O. When an object is written to a stream, information about its class must be stored along with the object. This class information is used to reconstruct the object once it has been sent across the communication path.

When using RMI, object serialization is used to send arguments of a method invocation from the client object to the remote object, and to send return values from the server object back to the client object. When using RMI, all classes that are to be serialized must implement the *Serializable* interface.[2] RMI also hides the details of the serialization from the client and server objects by letting the skeletons and stubs, the remote reference layer and the transport layer, Section 4.4.2, handle the matter[18].

4.3 System goals and advantages

This section describes the design goals of the RMI model. It states the items which were of most importance during the design phase[22] including *callback support* and *remote invocation*.

This section also describes some of the advantages (mobile behavior, object orientation), that emerged from the design.

4.3.1 System goals

The underlying design goal was to create *distributed object model support* for the Java language. That is, a model for using the Java language between JVM's and not just being limited to a single JVM.

The specification stated that Java object semantics were to be preserved and the safe environment maintained. In addition to this, there was also a wish to extend the language by introducing *callback support*.

[2]For a detailed description of the semantics of an interface please refer to: http://java.sun.com/docs/books/tutorial/java/interpack/interfaces.html.

Some of the important design goals are stated and described below, starting with the callback support item.

- Callback support, server-to-applet.

  The Java language allows downloading and execution of code (so called applets), these applets however do not interact with the server once they are downloaded to the client. RMI provides the necessary extentions to support this, letting the client make calls to the server and the server replying the request.

- Natural integration of the distributed object model into the Java language.

  The latest versions of the Java language includes RMI as an integrated component. This means that except for a few lines of additional coding nothing differs between writing RMI and non-RMI enabled programs. As all other packages RMI has its own classes, interfaces and methods.[3]

- Preserving the Java object semantics.

  The RMI package is as class- and object oriented as the rest of the Java language. Creating a remote object is no different from creating a local one, and there is no difference in using the objects.

- Maintaining the safe environment of the Java platform.

  The Java Virtual Machine is protecting its surroundings by using the Java *security managers* in combination with different *policies*. A policy grants permissions to programs and a security manager makes sure that the policies are followed.

  In RMI, safety is of most urge since objects are passed and downloaded between different JVM's. Since downloaded code actually can execute,[4] it must be limited to the current policy. This is accomplished using the RMI security managers.

[3]Refer to the API for details of your version of the Java language. http://www.java.sun.com.
[4]By being interpreted by the Host VM.

- Support remote invocation on objects in different JVM's.

  RMI provides the necessary classes for using methods located remote.

4.3.2 Advantages

Several advantages emerge from the design goals[22]. *Object orientation* is one, *mobile behavior* is another. This section states some of the "bonus" features emerging from the specification. Four of these items are stated below:

Object orientation: Java is based on object orientation. This object approach is maintained in RMI. RMI is (unlike RPC[5] which passes parameters but does not work well with program-level objects) able to pass entire objects as arguments. These objects are serialized and transported as bytestreams between the programs.

Mobile behavior: In RMI, the server program is responsible for implementing the service interfaces (see Section 4.4.2, for details). When the implementation changes (classes are changed and recompiled) the client that invokes one of the known methods automatically is answered with an updated object reflecting the changes. The client program does not have to be changed at all. All changes are made in the implementation class and the clients are implicitly updated. However, *and this is important*, as long as the implementation class does not add, remove or change arguments or types of its methods the client are unaffected of the changes. But, if one or more of these things happen the interface no longer matches the implementation and both the interface and the client must be changed and recompiled.

Security: Security in the system is provided by the existing Java security-manager which grant only the permissions explicitly stated by different policies. This makes an RMI application as safe as any other Java application.

[5]Remote Procedure Calls can be studied in depth on http://www.cs.cf.ac.uk/Dave/C/node33.html.

Simplicity: The additional lines of code needed to make use of the RMI technology are few. For example; to adapt a server to support RMI, only about three rows of additional code[27] need to be added to the existing server code. Furthermore all remote interfaces needed are ordinary Java interfaces extending the *java.rmi.Remote* interface. This makes the effort of writing RMI using programs fairly small.

4.4 The RMI model

This section describes, as mentioned earlier, *how* the process of transporting a method call between JVM's is carried out in RMI. This section is divided into three subsections treating the distributed application model, the architecture and the Naming service.

4.4.1 RMI distributed application model

A typical RMI application consists of a client and a server program. The server program creates 1..n remote objects and makes references to these objects visible (registering them). It then waits for connections. The client uses these references to invoke methods on the remote objects to achive the desired results. Remember that no additional code need to be installed on the client in order to adapt to changes in the server application.

The bytecodes (messages) are sent from client to server and from server to client by using any available URL protocol supported by Java, for example **HTTP**, **FTP** and **file**.[6]

Since the bytecodes are transported through the network there has to be a web server (a so called classloader) up and running to take care of the raw bytestream shipping between the clients and servers. Figure 4.4.1 shows a typical situation where a client and a server communicates using the RMI registry (Section 4.4.3) for service lookup and web servers for byte shipping.

[6]The file protocol can only be used if both client and server have access to the same filesystem.

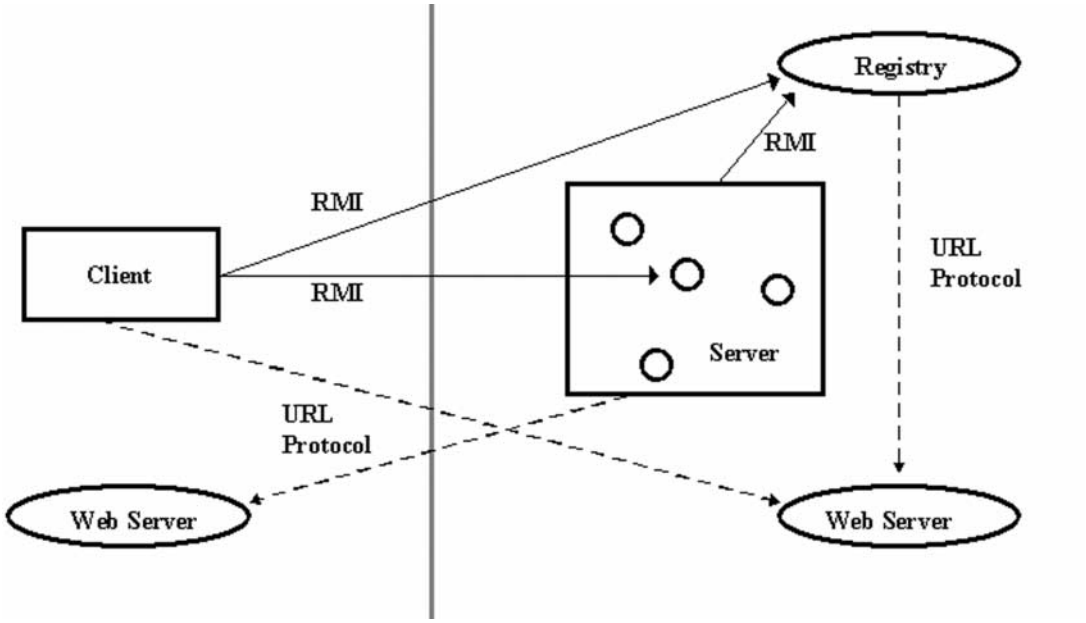

Figure 4.4.1: A distributed object application using the RMI registry

4.4.2 RMI architecture

The RMI architecture is based on the principle that definition and implementation of behavior are separate concepts[5]. The definition is coded in an *interface* file and the implementation is coded in a *class* file. The class file implementing the behavior is located on the server and *only* on the server. The definition file (the interface) however must be present to both the client and the server.

"Interfaces define behavior and classes define implementation"[5].

The least that is needed in order to run an RMI application are three files. One interface file and two class files which both implements the interface. One of these files resides permanently on the server side and the other (known as the proxy object, or *stub*) is during run-time sent to the client before the method call is done.

The interface defines 1..n methods that can be remotly invoked. Each of these method definitions defines exactly one service. The interface file is used by both the client and the server program and must be located at both the client side and the server side. This is

necessary because in order to get the RMI communication to work, both the proxy object class and the implementation class must have access to the method definitions located in the interface file.

As seen in Figure 4.4.2 the client uses the proxy object to make method calls to methods that the server implements. The communication is based on the common interface.

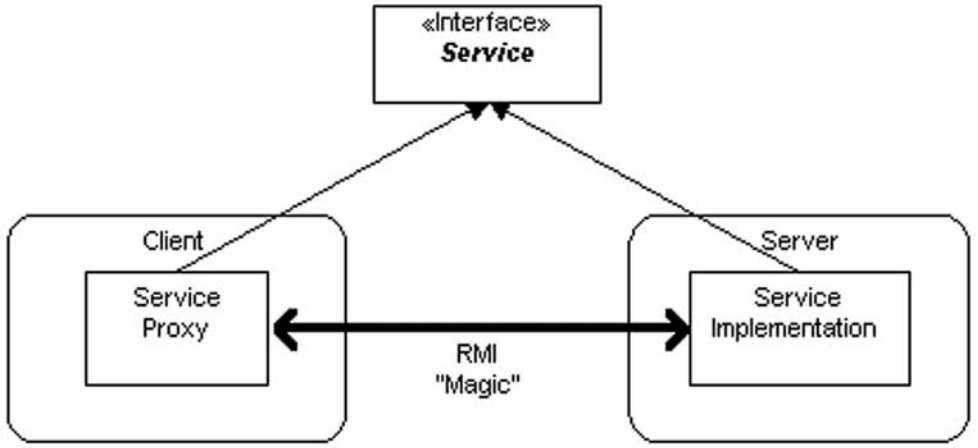

Figure 4.4.2: A common interface which is implemented by both the client side and the server side

Architecture layers

The above section provides an overview of the RMI architecture. This section describes the more detailed layered approach. RMI architecture contains three layers[27]. The *Stub and Skeleton* layer, the *Remote Reference Layer (RRL)* and the *transport* layer. Figure 4.4.3 shows the layered architecture of the RMI system.

Stub and Skeleton layer: The stub and skeleton classes are both generated from the implementation class with the *rmic* command.[7]

When the client program makes the call to the server, the stub class is serialized and sent to the client. The stub then intercepts the method call made by the client and

[7]The rmic is the RMI Compiler program. It creates skeleton and stub classes. Further information about rmic can be found on http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/rmic.html.

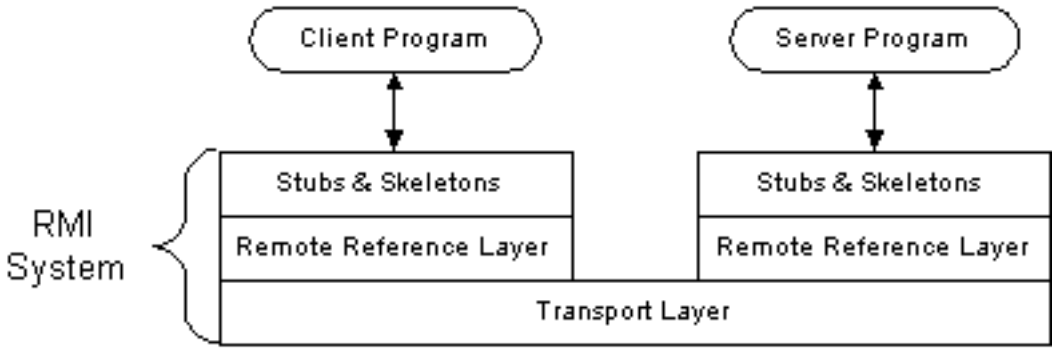

Figure 4.4.3: The layered architecture

redirects it to the remote RMI service where the skeleton class, which knows how to communicate with the stub over the RMI link, receives the call and forwards it to the server program.

So, what actually happens when a client makes a remote method invocation is that the method call goes from the client program to the stub, from the stub over the RMI link to the skeleton that forwards it to the server program. The server answers the call by serializing an instance of the implementation class and sending it to the client.

Remote Reference Layer: The RRL defines and supports the semantics of the RMI connection. This layer provides a representation of the link between the client and the server.

Transport layer: The transport layer is responsible for the connections between Java Runtime Environments (JRE's), it is also designed to make connections in the case network problems arise (which makes the connection reliable).

Figure 4.4.4 shows the connection between JRE's. The transport layer is in the bottom of the JRE block, just above the Host OS layer. All RMI connections are stream-based TCP/IP[23] connections and this requires an operational TCP/IP stack on the machines that run the RMI protocol.

On top of the TCP/IP stack Java Remote Method Protocol resides. This is a stream based protocol.

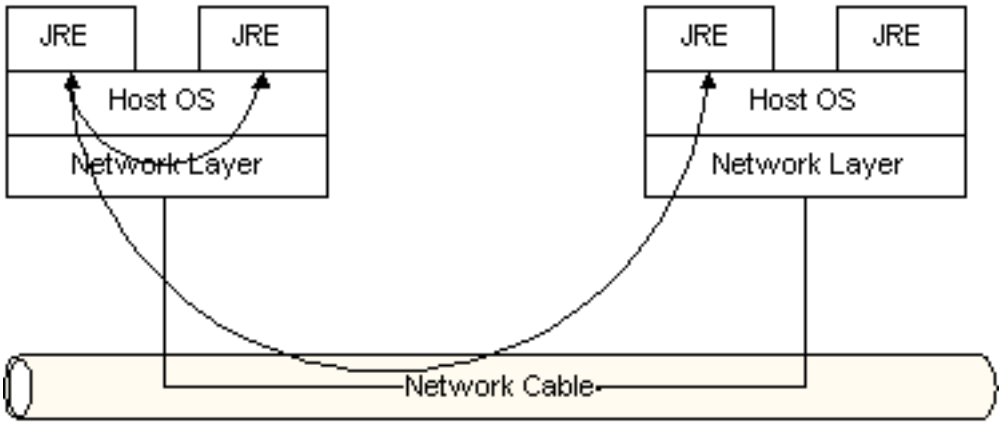

Figure 4.4.4: The connection between different JRE's.

4.4.3 Exporting services and Naming

When a service is started in the network it is exported to a remote registry server. By doing so the service provider (the server on which the service resides) grants the clients access to the service. The remote registry server can be thought of as a database containing mappings between names and remote objects. The heart of the remote registry server is the *object* registry which performs the mapping.

An arbitrary registry object can be placed in two specific locations:

- In the same JVM as other server classes.
- As a standalone registry server on a separate JVM.

After a service is registered by the server, the clients must first obtain a reference (also known as a stub) to the remote object in order to use it.

Both the referencing and the registering process are handled by using the *naming service*.

The naming service depends on the *java.rmi.naming* class[8] that provides methods for storing and obtaining references to remote objects in a remote object registry. Two methods of the class are important for the referencing and registring, namely:

bind: is used when a server want to bind a name of a service to a specific object and register the reference.

lookup: is used when a client wants to obtain a reference (downloading a stub).

Both these methods take an argument that is a URL on the form:

rmi://host:port/name

where rmi is the rmi protocol, host is the host on which the registry is located, the port is the port used (default 1099) and the name is the name of the service.

4.5 Conclusion of the RMI technology

With a brief repetition this section concludes the RMI introduction. Important to remember is the following:

- RMI is short for Remote Method Invocation.
- RMI is a model for distributed computing using Java objects.
- RMI is naturally integrated in the Java language.
- RMI is used for communication between JVM's.
- RMI has a three level layered architecture.
- The server provides the services by implementing the service interface and registering services with the registry.

[8]Refer to the API for details.

- The client requests the services by looking them up at the registry.
- The naming service is used to bind services to specific names and to look them up.

Chapter 5

Jini

This chapter is focused on providing a short introduction to the Jini technology.

The first section, "What is Jini?", defines the concept, explains what a *service* is and what environments Jini is particularly well suited for. Section two, "The Jini design goals", goes in-depth with the goals of the Jini design. It covers simplicity, reliability, scalability and support for various devices.

Section three, "Jini architecture", describes the layered Jini architecture model, the lookup service and the environmental demands. It also gives examples on different device architecture models.

The last section, "The basic Jini concepts", describes the five Jini concepts that the whole technology relies on, *the heart of Jini*.

Figure 5.0.1: The Jini logotype

5.1 What is Jini?

Jini technology is an infrastructure designed to provide services in a network and to create spontaneous interactions between applications using these services. A service can be thought of as the capability of printing a document, scanning in a picture, changing channel on your television set, using the play button on the stereo or turn on the kitchen lights.

Services can join and depart the network in a robust fashion. This means that joining and departing are safe operations that can be controlled to avoid making the system unstable. The clients in the network can rely on that all services visible to them are available or at least that the reason why a service is not available (failure) is clear and that it does not endanger the clients operations.

A Jini community adapts automatically when services come and go. In addition, no device drivers need to be explicitly installed on the clients in the network in order to use the different services (no human administration is needed). Interaction between clients and services are accomplished by the use of Java objects provided by the services. These objects are loaded into the clients application and are used as an interface[1] for the communication between the client and the service. Note that the client easily can use the service without having seen it before, ever.

This is the Jini technology in a nutshell. These special features makes Jini suitable for environments where mobile computing devices and software based services come and go dynamically. Figure 5.0.1 shows the Jini logotype.

5.2 The Jini design goals

The basic design concept behind the development of the Jini technology is to allow for services to be available to any client that can reach them. This should also be done in a type-safe and robust way. That is, a Jini enabled device is able to use services from

[1]Not to be confused with the Java language interface concept.

all providers in the network as if the device was directly attached to them. The four key concepts[4], i.e. the areas that the designers felt were the most important, of Jini are listed below:

- Simplicity
- Reliability
- Scalability
- Support for various devices

5.2.1 Simplicity

Fundamental Java technology[2] is the basis of all Jini building. In addition to Java, Jini adds only a thin layer allowing network devices and services to easier work together. That is, knowing Jini is not very different from knowing Java. The programmer does not actually need to learn anything new except the Jini API.[3] Basically, Jini is designed to handle *how* services connect, not what kind of services they are, not how they work and not what they do. An arbitrary Jini service does actually not even have to be written in Java. The only thing required is that somewhere in the network there exists some Java code that can be used to handle the client-service lookup and connection (see Section 5.3.2 for details on the lookup service). From the Jini perspective everything connected to the network is cathegorized as services. A printer or a scanner can thus be treated as services.

5.2.2 Reliability

An arbitrary Jini community is virtually free of administration. Almost no human interaction is needed for a community to maintain a stable state. This is the result of the

[2]This is described in great detail in the Java Language Specification. Refer to http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

[3]The Jini API is part of the downloadable Jini1.1 beta 2 package from http://developer.java.sun.com/developer/products/jini/EAproduct.offerings.html.

spontaneous networking and *self healing* abilities of Jini. Below is a brief explanation of the terms spontaneous networking and self healing.

Spontaneous networking: Spontaneous networking is the ability for services close to each other to automatically form communities without external aid. Whenever a service is connected to the system it is immediately ready to use. No configuration of the system (editing config files or configuring gateways) need to be done and the result is that the set of services can dynamically be changed during execution of the system. The set of available services grows and shrinks with time and interested participants are automatically notified whenever there is a change in the set.

Self healing: Jini is constructed on the basis that all systems have a certain probability of failure. Because of this, a Jini system will in time repair damage inflicted on itself. The concept of leasing allows the system to exclude a non working member of the network automatically. If the device crashes, its leases expires and the system does not need to take further notice of the device (leases are covered in Section 5.4.3).

5.2.3 Scalability

Scalability is achieved by letting Jini communities join together to form federations. A *Jini community* is a limited set of services that are all aware of each other. A *federation* is a set of communities that have knowledge of each other via the participating communities lookup services. This feature makes a large system easy to handle since each participating community is only a limited set of services even though the system as a whole may be large.

5.2.4 Support for various devices

Jini is designed to support many different entities participating in a community. An entity is a piece of hardware, software or a combination of both. It is not of any importance

for a client to know what type of entity holds the requested service. It is enough just to understand the interface it presents. In case the entity is a hardware component, Jini is very flexible about the amount of computing power this entity is capable of. A hardware entity can be a full fledged stationary workstation with a full version JVM[4] as well as a simple office lamp containing virtually no computing power. The latter case does however require some sort of computational device connected to the network by a Jini interface and connected to the lamp by a specific lamp interface. This device is commonly called a *proxy*. An additional feature is that a service or device does not actually have to be written in Java, as long as a proxy between the network and the device is capable of understanding both languages.

5.3 Jini architecture

Jini is a distributed computing framework[26]. Each participant in a Jini community either offers services, uses them or both. Traditionally a server provides services and a client is using them. In Jini, however, there is a slight difference. Every participant in a Jini community can act as both client and server. This section describes the basic architecture model, it explains how the lookup service works and the environmental demands which have to be satisfied by a Jini community. At the end of the section there are examples on different device architecture modeling (ways to set a Jini-network up).

5.3.1 Basic Architecture model

As seen in Figure 5.3.1, the client and service applications are on top of the hierarchy. Below is the Jini layer which provides the classes necessary for the communication (leasing, discovery, lookup). The Java layer includes the RMI-classes[5] necessary for the remote

[4]The JVM specification is available from http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html.
[5]RMI is treated separately in this document.

interaction as well as all socket-handling classes, and everything else. The Java VM in which all Jini and Java classes resides, lies on top of the host OS which in turn uses the transport layer to ship the data across the network (usually using the TCP/IP protocol[24]).

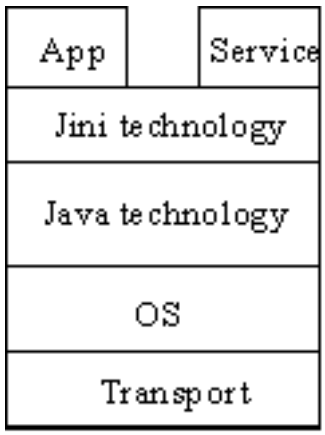

Figure 5.3.1: This picture shows the layered Jini model. The application takes advantage of Jini which in turn depends on Java.

5.3.2 Finding services

Jini is built around the lookup service[4]. The lookup service is what makes the clients find the requested services (note that the lookup itself is a service, just like any other service in the network). Every Jini community contains one or more lookup services (the more there are, the safer the system gets). Every time a service is booted on the network the discovery process is invoked. This process is used by the service to find the local lookup service (Section 5.4.2). Next, the service registers its proxy object (Figure 5.3.2) which is a Java object including the interfaces the service implements. The interface concept is crucial to Jini because it is through the interfaces the communication takes place.

Whenever a client whishes to use a service it asks the lookup service for it. As described later, this question can be asked in multiple ways, depending on the information the client

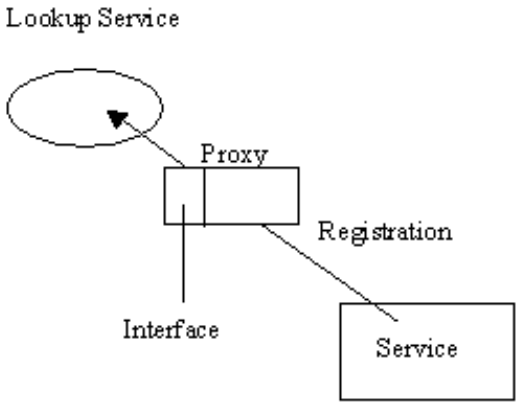

Figure 5.3.2: The newly connected service registers its proxy with the lookup service. The proxy object's interface is used to define the service.

has. The lookup service then returns the proxy object to the client and the client downloads the code for the object. Refer to Figure 5.3.3 for details.

Once the proxy object is downloaded the client can use the service simply by invoking methods on the proxy object which in turn communicates with the service to execute the requests. This is shown in Figure 5.3.4.

5.3.3 Environmental demands

There are certain requirements that have to be fulfilled in order to get the Jini technology working. The most important prerequisite is that there has to be a network. Also, the network must support multicasting (today through TCP/IP[23]). The network must contain Java with included RMI support[6] since the services must be run on a JVM or having a JVM as a proxy. The clients and services must be able to publish/retrieve proxy objects at the lookup service. The clients must know what type of service they are looking for, at least at a minimal level of detail.

[6]It is possible but not recommended to write a substitute for RMI to avoid using the RMI classes.

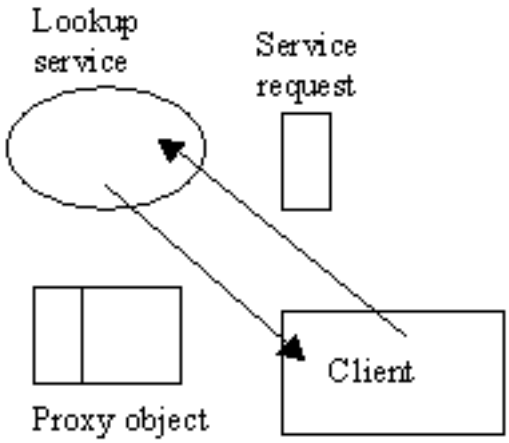

Figure 5.3.3: The clients request is answered by the lookup service by sending the requested services proxy object back to the client.

5.3.4 JVM dependencies

The Jini architecture rely on certain abilities of the Java VM to run properly. These abilities includes the following: Ensurance that the code behave the same everywhere, serialization of Java objects for transportation, virus protection and safety during the transport.

5.3.5 Device architecture modeling

Jini services can be implemented in serveral types of hardware[26]. Listed below are four of these types as well as their advantages and disadvantages:

Resident Java VM devices

A device including computational power, memory and secondary storage enough to host a full JVM and Jini infrastructure support (typically a PC). Having a full JVM locally gives services the advantage of using RMI to communicate between the client and the service host once the service proxy has been downloaded into the client. In addition virtually all

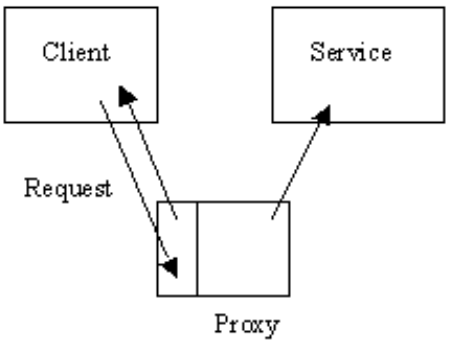

Figure 5.3.4: The client request a service by invoking methods on the proxy, which in turn communicates with the service.

downloaded code will run. This approach is simple and flexible but it is also expensive because the need for a local processor, memory and permanent storage.

A resident Java VM architecture example can be seen in Figure 5.3.5.

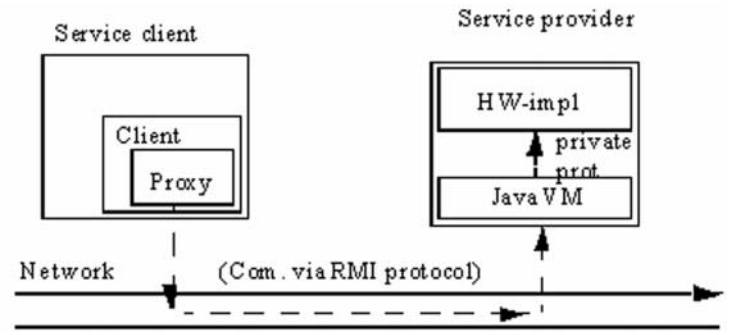

Figure 5.3.5: Communication over RMI between machines with resident JVM's

Specialized VM devices

A device which has not a full JVM installed can still participate in a Jini community, but can not use all the Jini flexibility. A specialized VM must at least support the Discovery protocol and the Lookup service. These features must be implemented in the device by

whomever manufactures the device. An advantage is that a reduced JVM results in less memory needed, which makes the implementation suitable for devices with little computing power and memory resources. Unfortunately it also reduces flexibility and the device becomes tied to the particular implementation of the Lookup service that the specialized VM supports.

Clustering devices with shared VM (physical option)

This approach lets several devices use the same JVM. This distributes the cost[7] over the devices. The Java device bay can be thought of as a provider of power, network connection and a processor. The interface between the different devices and the JVM is arbitrary. Figure 5.3.6 shows an example of the model. This approach is positive because it is cheaper to only have one device containing the JVM, but it also brings redundancy to the system because a protocol between the devices and the device bay must be defined in advance.

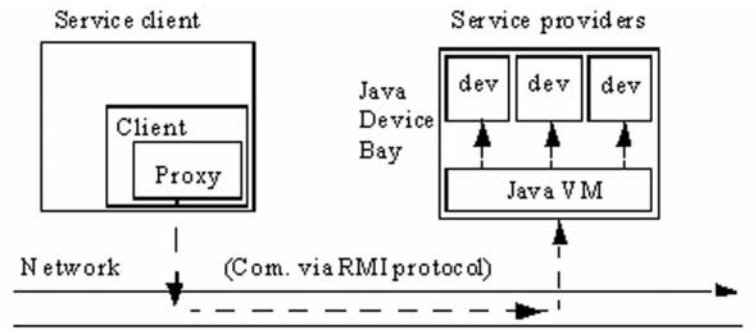

Figure 5.3.6: Devices physically sharing the same JVM

Clustering devices with shared VM(network option)

This approach is basically the same as the physical option above. But the location of the devices differ. Here, the devices are connected somewhere in the network using the JVM

[7]The cost for the processor and memory.

entity as a proxy. In this way the devices do not need physical contact with the proxy. The devices use some private protocol when talking to the proxy, who in turn uses the Jini protocols when talking to clients. This is pictured in Figure 5.3.7. Advantages includes the following: no physical constraints limits the number of devices that can be connected to each proxy and each device does not need its own computing power. Disadvantages includes: extra hardware (power supply, network connection) is needed for each device and extra protocols are needed for device identification at the proxy.

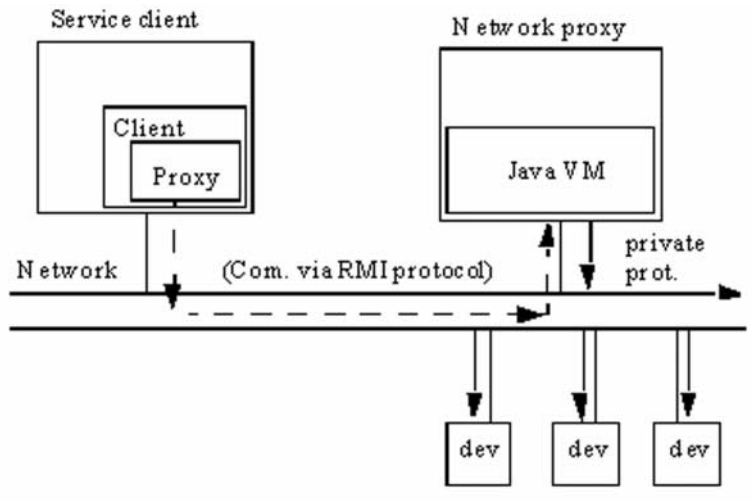

Figure 5.3.7: Devices sharing the same JVM over a network

5.4 The basic Jini concepts

There are five basic concepts that constitutes the Jini services ability to self-heal and to support spontaneously created communities[4]. In fact, these concepts are all a user need to know and understand to use Jini. These concepts describe discovery and joining, lookup, leasing, remote event handling and transactions. This section will briefly explain each of these concepts.

5.4.1 Discovery and joining

The procedure to find a Jini community is known as discovery. During this procedure all available lookup services are found. There are three different discovery protocols depending on the situation and the need:

The Multicast Request Protocol: The Multicast Request Protocol is used by a newly booted service that wishes to find all available lookup services in the area.

The Multicast Announcement Protocol: The Multicast Announcement Protocol is used by lookup services when they want to tell the community that they are up and running.

The Unicast Discovery Protocol: The Unicast Discovery Protocol is used when the exact location of a lookup service is known and a client wishes to talk directly to it. It is also used when static links between lookup services need to be created, as when two or more lookup services are to be federated together.

In addition to the discovery protocols which are used to find the lookup services there is also a protocol called the join protocol. This protocol is used when an entity gets to be a part of the community after discovery is done. When an entity has joined the community it can use the lookup service to publish its service items (full description of the service) and to take advantage of the services offered by the community.

5.4.2 Lookup

Lookup is the process of using the available lookup services, i.e. asking for services. An entity requesting a service can do this in one of the following ways:

Proxy-type searching: The client searches for the type of the proxy object in the service item.

ServiceID searching: Every service is assigned a (UUID), a Universally Unique IDentifier (128 bit value). If the client explicitly knows this number the search can be done this way.

Attribute search: The client can search the attribute list included in every service item.

Below is a skeleton for a ServiceItem class. All members of the class (serviceID, service and attributeSets) can be used as search parameters.

```
public class ServiceItem implements Serializable{
        public ServiceItem(ServiceID serviceID, Object service,Entry[]
        attributeSets){
        ...
        }
        public ServiceID serviceID;
        public Object service;
        public Entry[] attributeSets;
}
```

5.4.3 Leasing

Leasing is the concept of granting the Jini community stablilty as well as self-healing abilities in the case of a system crash or a network failure. Leasing means that the resource (service) is leased to an requesting entity for a pre-determined period of time. Some properties of the leasing concept are stated below:

- Leases may be denied
- Leases can be renewed
- Leases expire after a pre-determined period of time

The advantage of having resources leased is that after some time all unused resources are freed because the lease time expires. This means that virtually no system administration is needed.

5.4.4 Remote events

The remote event handling system of Jini is used for asynchronous notification. An entity that depends on the state of another entity in the network is most likely interested in being informed when the state changes. This is accomplished through the remote event mechanism.

5.4.5 Transactions

Transactions in Jini is carried out using a modified version of the two-phase-commit protocol. The participants in a transaction implement the TransactionParticipant interface and the Jini transaction manager run the 2PC[2] protocol for them.

5.5 Conclusion of the Jini technology

This section concludes the Jini introduction with a short listing of the most important features previously explained.

- Jini technology is an infrastructure designed to create spontaneous interactions between applications in a network.
- Jini design goals are: simplicity, reliability, scalability and support for various devices.
- Jini is designed to handle *how* services connect.
- A Jini community is a set of Jini services.
- The lookupservice is the heart of Jini, keeping track of all services available.

- Four ways of modeling the device architecture are: resident Java VM devices, specialized VM devices, clustering device with shared VM (physical option) and clustering device with shared VM (network option).
- Discovery and joining, lookup, leasing, remote events, and transactions are the five basic Jini concepts.

Chapter 6

Jini in Bluetooth networks

This chapter treats the basic concept of the combination of the Jini and Bluetooth technologies with a simple example (much like the application developed and described in the next chapter). In addition, the configuration of the existing Bluetooth link as well as the result of the configuration is presented in this chapter.

The Jini technology implicitly installs the device drivers necessary and the Bluetooth technology makes cables between devices unnecessary. These two concepts makes a very interesting usage model when combined. There are two key features in this usage model:

Ad hoc networking: The ability to make spontaneous networks with Bluetooth devices is a key feature within wireless networking. Devices equipped with Bluetooth interfaces connect to other devices in the proximity[1] and form ad hoc networks. These units can simultaneously communicate with up to seven other units and form piconets. The Piconets can easily connect to a LAN or some other kind of network. The Bluetooth technology replaces cables in networks and the units can use the network as any kind of network. The focus in this work is on TCP/IP networks and the possibilities with Jini in Bluetooth networks.

[1] See Table 3.5.2 for further reference to the different power classes.

Automatic configuration: Dynamic service allocation is a relatively new idea in networks. Devices that are plugged into network self-configure, devices can be a printer, a mouse, a remote control, etc. With the use of TCP/IP and the use of Jini applications, the networks will become operating system independent[1]. There are lots of different technologies of making self-configuring networks, the focus in this work is on the Jini technology.

All of the Bluetooth and Jini specific features are explained in their respective chapters in this document.

6.1 Combining the two concepts

Jini is especially well suited for ad hoc networking because of the ability to register/unregister services in a spontaneous way, much like how Bluetooth works with devices that appear and disappear in the network. Leasing in Jini controls the services in the network and always keeps the lookup service up to date. The combined concept of Jini and Bluetooth is simplest described with an example. Take for instance a remote control (the application model used in this work) that is "dumb", not knowing anything of what it is supposed to control. This remote control is in need of a device driver to whatever it is to control.

Figure 6.1.1: A dumb remote control.

Figure 6.1.1 illustrates a remote control not knowing anything about its surroundings and what it is to control. This can be called a dumb remote control. The remote control is ready and listening for new interface to services that is remote controlable. We have a generic remote control.

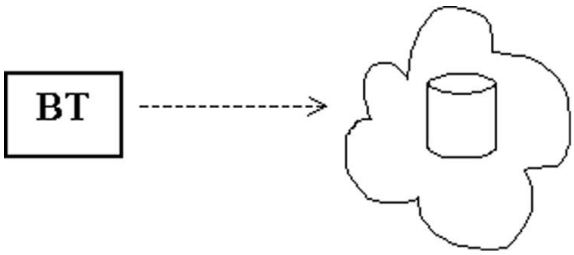

Figure 6.1.2: A remote control located in a network.

Figure 6.1.2 illustrates a Bluetooth enabled remote control just being introduced to a network. This network is also Bluetooth enabled and has TCP/IP support. These features makes communication between the device and the network easy. The network contains a database with the available services, the lookup service, in the network. The remote control is introduced to the Bluetooth network through paging/inquiry[2] when the device is within proximity of the network. In the Bluetooth specification the SDP protocol[3] will have some features for discovering devices and enable the use of some kind of ad hoc protocol such as Jini in the future.

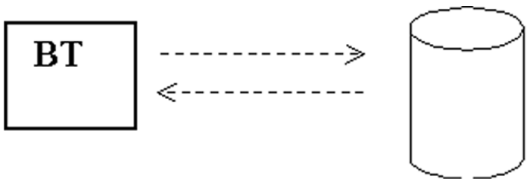

Figure 6.1.3: The remote control talking to the lookup service.

Figure 6.1.3 illustrates the remote control as a member of the Bluetooth network talking Jini to the lookup service and asking for the available services in the network. The lookup service keeps a dynamic list of all the services that are running on the network. A service can be a printer, a television set, a CD-player, etc. All of these services send their respective drivers to the service user (in this case the remote control) when asked for.[4] When the

[2]See Section 3.5.2 (channel control) for further information on paging/inquiry.
[3]See Section 3.5.7 for further reading.
[4]Only the services that match the remote control in this case, see the Jini chapter for more information on this topic.

services is no longer available the network is being alerted by the expiration of the lease, this is a Jini specific feature. This way of communicating and keeping services up to date is truely ad hoc management of the network services. These features makes Jini in Bluetooth networks a very good combination.

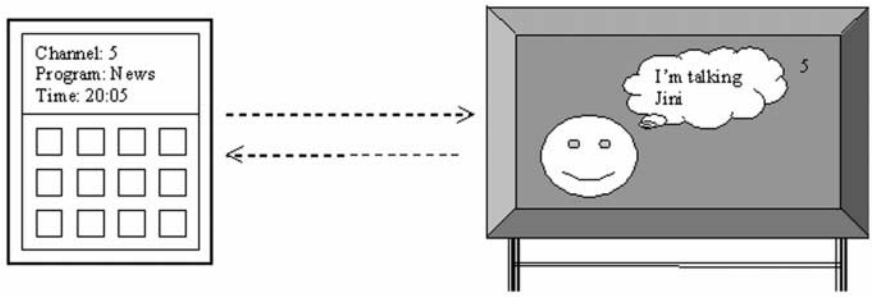

Figure 6.1.4: The remote control talking to some service in a "old fashion" way.

Figure 6.1.4 illustrates the remote control talking to a real application, in this example a television set. The TV and the remote control communicate wireless using Bluetooth and its radio link in the ISM frequency band, this is the physical aspect of the communication. On the application level the remote control and the TV talk some kind of language the TV provides the remote control with. The language may contain primitives for changing channel, volume in the remote-to-television communication way and information about the time and channel from the television to the remote control. This duplex communication between the remote control and television set is enabled through the Bluetooth link between the devices not like the simplex communication of the old fashion infra-red remote control. The language is a part of the driver the TV sends to the remote control, the graphical interface is another part of the driver. Now the "dumb" remote control has become a television set remote control that knows all the necessary information about the TV and its features and is also able to receive information from the TV. This same remote control can also control the CD-player or the Video for instance, it is all about the interface the service provides to the remote control.

6.2 Configuration of the Bluetooth (wireless) link

The hardware consisted of two Digianswer developers kit[3]. These consisted of one democard (PC-card), one RS232-adapter (serialcard) some testapplications, drivers and manuals. The democard supplied the computer with a regular Ethernet connection. The RS232-adapter supplied the computer with a virtual serialport. The basic configuration of the two cards was different and this resulted in some problems further described below. The developers kit for the RS232-adapter was not the latest version of the hardware and this implied the following problems:

Different versions of the drivers for the democard and the RS232-adapter: This forced us to downgrade the driver of the democard since no later version of the RS232-adapter was available. This was a problem since not all of the testapplications were present in the old version.

Only the democard supported direct PPP-over-Bluetooth: This was what we needed for the project. The ability to access the TCP/IP network is essential in this work.[5]

Time consuming configuration: The configuration of the link was too time consuming and complex for the project. This lead to the use of existing 802.11b (WLAN)[20] wireless equipment owned by Fyrplus AB.

These problems could probably be corrected by upgrading the hardware to the lastest version of the developers kit and by using two PC-card kits. The upgrading of the hardware was not worth its price since the availability of Bluetooth access points to a more reasonable. The result of this configuration lead to the use of a WLAN connection for the Jini application. The application will run in exactly the same way with this kind of network as with a Bluetooth network.

[5]See the Jini chapter for further reading.

Chapter 7

Application design

The ideas behind the use case and the design of the application are further explained in this chapter. Concepts from the Jini and RMI section are used without any further explanation.

The running application, as seen in Figure 7.0.1, is made up by a serverside program (the jukebox) with the ability to play music (Mp3-files[1]), a number of client programs (remote controls) which will control the server program and a lookup service providing support for the Jini service. The network in which the application will be running is a TCP/IP network.

A more thorough description of how the communication works, with references to the Figure 7.0.1 is given here:

1. The jukebox publishes a proxy object of the service it owns, i.e. the ability to play mp3-files in a FIFO order (just like the jukebox in a restaurant), to the network and all its available lookupservices. Sun's lookup service Reggie is used.

2. The client program, UniversalRemoteControl, asks the lookup service what relevant[2] services the network can offer.

[1] Mp3 is a popular music file format.
[2] In this case only services of type RemoteControl.class are considered relevant.

3. The lookup service answers the clients program with copies of the proxy objects that match the question.

4. All requests by the clients go to the same service, i.e. the jukebox. This means that a change in the song queue by display one for example will affect the jukebox and all the client programs, i.e. the different displays. The request goes through the Jini network (4) to the jukebox server (5), this server updates all the connected proxy objects through their respective "link" (4) and all changes in the jukebox will be updated in a uniform way to all the displays.

5. A change in the play list occurs when the queue of playing songs change, for instance when the first song in the queue is finished playing. Updating through (5) and (4) are done like in the previous point.

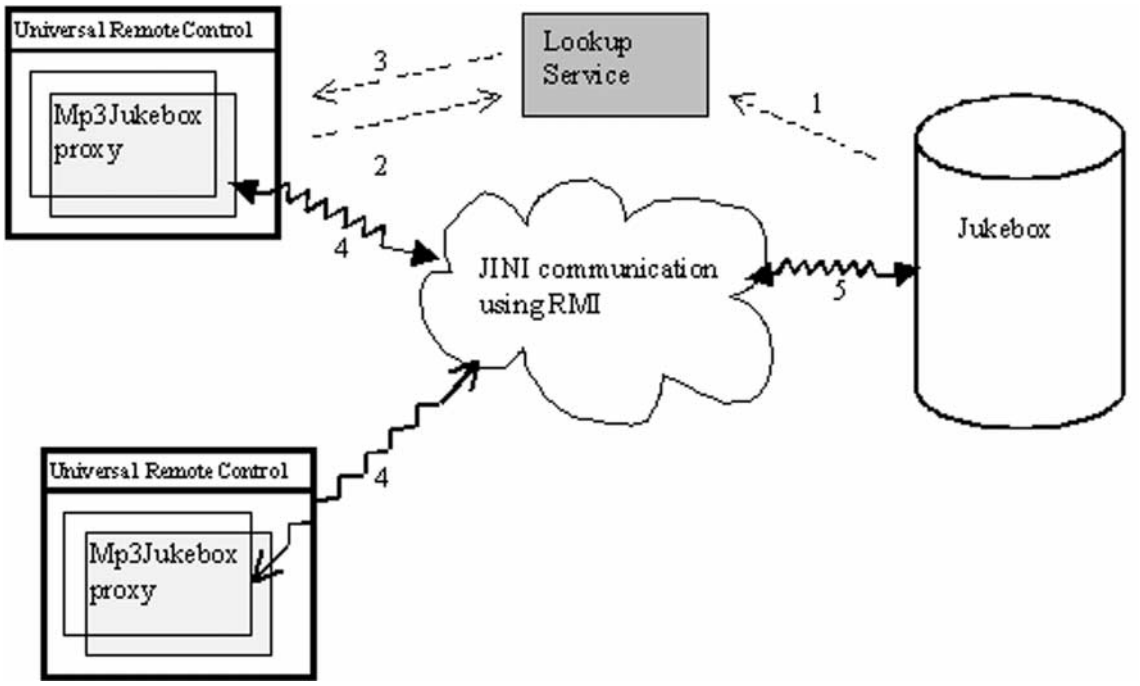

Figure 7.0.1: The process of communicating in the Jini network.

The design phase can be decomposed into five steps: *The RemoteControl Interface*, *The*

client design, *the server design*, the *service design* and the *class-generation using UML*.

7.1 RemoteControl Interface

The RemoteControl interface consists of the following code and methods:

```
import net.jini.core.lease.*;

public interface RemoteControl {

    public String getName();

    public java.awt.Component getDisplay();

    public Lease lease(long time)
throws
java.rmi.RemoteException,
LeaseDeniedException,
UnknownLeaseException;
}
```

There are some things that are important to know about a remote control and these are methods in the interface. The following methods are used:

getName(): The UniversalRemoteControl needs to know the name of the service to be able to show the right name of the service in the choice box of the graphical interface of the UniversalRemoteControl as can be seen in Figure 8.1.1.

getDisplay(): The UniversalRemoteControl also needs a graphical interface to be able

to control the different services which already have been discovered. In this use case the control of the Mp3JukeBox.

lease(long time): The UniversalRemoteControl finally needs to know for how long it is permitted to use the service (how long the lease time is).

7.2 Client design

The user of services, the client, should only be able to do a limited set of operations thus letting the server do all the work. Since a client using few resources can be run on devices with slow processors and little memory it is suitable for handheld devices such as the PDA or a cellular phone. There are however some operations that the client must support in order to participate. These are:

- The ability of discovering services already registered on the network, i.e. asking the lookup service about the available services.
- The ability to be informed when a service is made available on the network, i.e. receiving proxy objects from the lookup service after making the request.
- The ability of leasing the servers resources for an arbitrary time (not forever). This is accomplished through the lease manager implemented in the client.
- The ability of receiving proxy objects and displaying these to the user via an interface. In this case as a Java Panel in the UniversalRemoteControl.

The interface of the client should show the different services that the network offers. The client program, UniversalRemoteControl, receives *RemoteControl interfaces* from the lookup service and these should be chosable and showable in the display. The queue of the songs in the current "playing list", which the jukebox owns, should be dynamically updated in the display. The list of all the songs which the user can choose between should

be visible and, finally, some way of submitting the chosen songs to the jukebox should be implemented.

7.3 Server design

Since the client does not have much functionality some additional features must be added to the servers responsibility. Because of this, the server must contain much more features. The server must support at least these operations:

- Providing a service, i.e. the proxy object.
- Discover the lookup service and publishing proxy objects on the network.
- Write serviceID (a unique id for every proxy) objects to file.
- Read serviceID objects from file.
- Grant leases to clients wanting to use the service using a landlord.[3]
- Renew leases.
- Disconnect clients whose lease have expired.

The above stated items are the core in the application. These items constitutes the basic communication features needed to build a Jini-application. The server should be able to load directories and find all the mp3-files in these. The server is the physical jukebox and should have some kind of hardware for playing mp3-audiofiles.

7.4 Service design

In addition to this mandatory features the server needs to have a service for the client to download (i.e. a proxy object). The goal of making a use case is to produce a simple and

[3]The entity owning the service proxy and also administers it.

clear demonstration of the Jini and Bluetooth technologies. One way of realising this is to include both visual and audible features. An appropriate choice was an jukebox using the mp3 format, a mp3-jukebox. The necessary features supported by the service are listed below:

- Displaying an interactive interface at the clientside.
- Playing songs chosen by the client.
- Dynamically update all clients when the state of the player changes, i.e. when songs are added and removed.

7.5 Class-generation using UML

Both the client and the server side contains a main method and a runnable thread preventing them from terminating while executing. The Player capabilities are all residing on the server program since the server supplies the service. The final version of client side UML[17] design can be viewed in Figure 7.5.1 and the final version of the server side can be viewed in Figures 7.5.2 and 7.5.3.

This is a brief description of the classes used in the applications. The Java classes provided by the development environment are *not* described here. Relationships are shown in the UML diagrams and are not described here. The client side contains the following classes:

- The main class on the client side is the **UniversalRemoteControl** class. It contains three listener classes.
- The **Listener** class listens for new lookupservices on the network at client startup time.

- The **ServiceEventListener** class listens for lookupservices available on the network after the client has started.
- The **LeaseList** class listens for events regarding the service lease.
- The **RemoteControlFrame** class creates the client application GUI[4] and also fills it with the matching discovered services. It contains awt[5] components for the building.

The server side contains the following classes:

- The server side main class is the **Mp3Jukebox** class. It contains a listener.
- The **IDListener** class obtains the unique serviceid the service gets the first time it registers its proxy.
- The **Mp3Player** class is responsible for playing the songs.
- The **Mp3Display** is responsible for the appearance of the service in the client applications interface. It also contains three listener classes.
- The **DirListener** class listens on new additions to the *songlist*.
- The **SongListener** class listens to mouseclicks on the SongList and changes the label to a textfield.
- The **Mp3DisplayListener** class listens to changes to the play- and songlists.
- The **SongList** class finds all mp3files in the input directory.
- The **ServerLandlord** class keeps track on all leases clients have on the service. It also renews the leases.
- The **Mp3RemoteControl** class is the proxy object class.

[4]Graphical User Interface.
[5]Abstract Window Toolkit.

Figure 7.5.1: The client side of the application design.

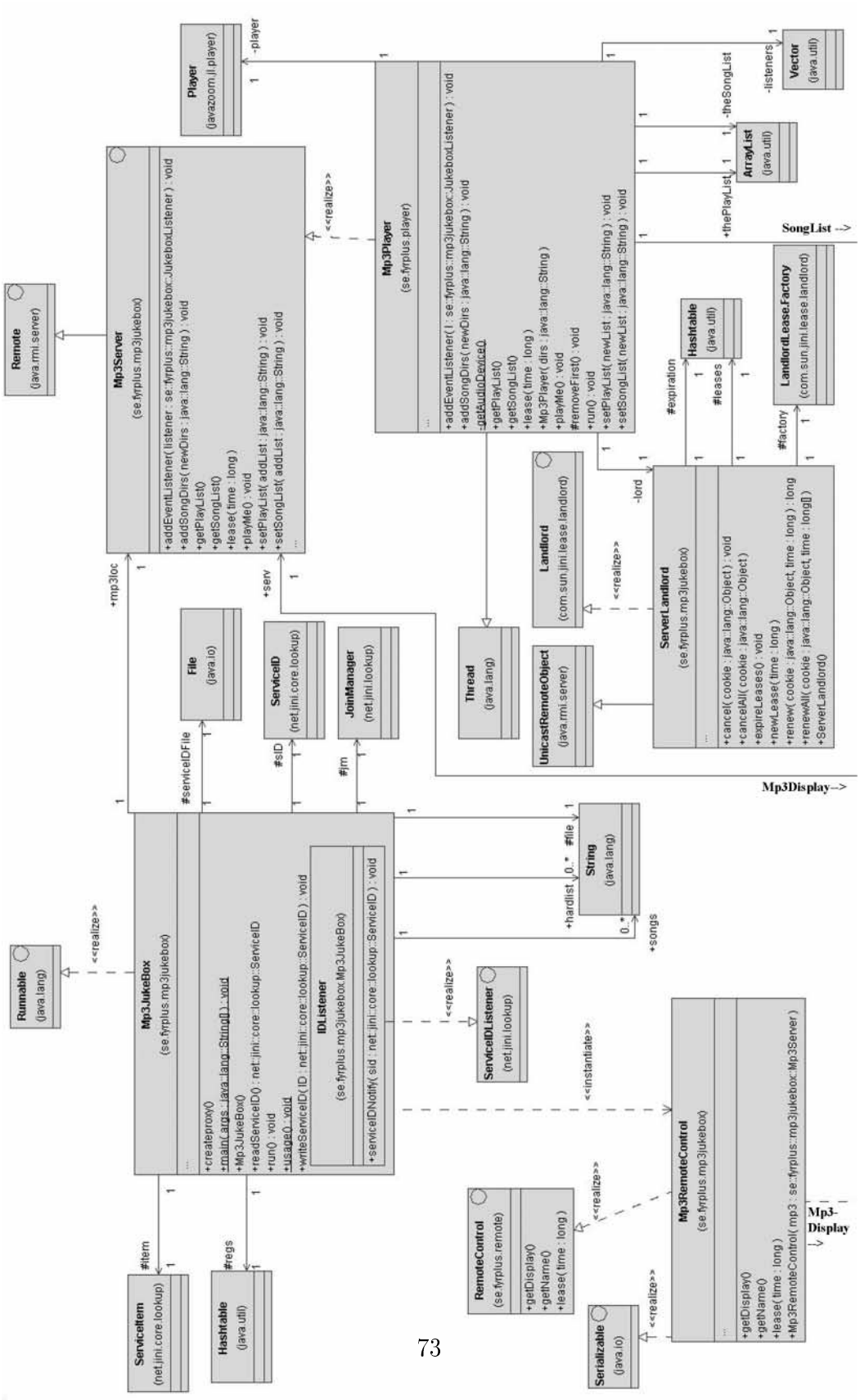

Figure 7.5.2: The server side of the application design, part 1.

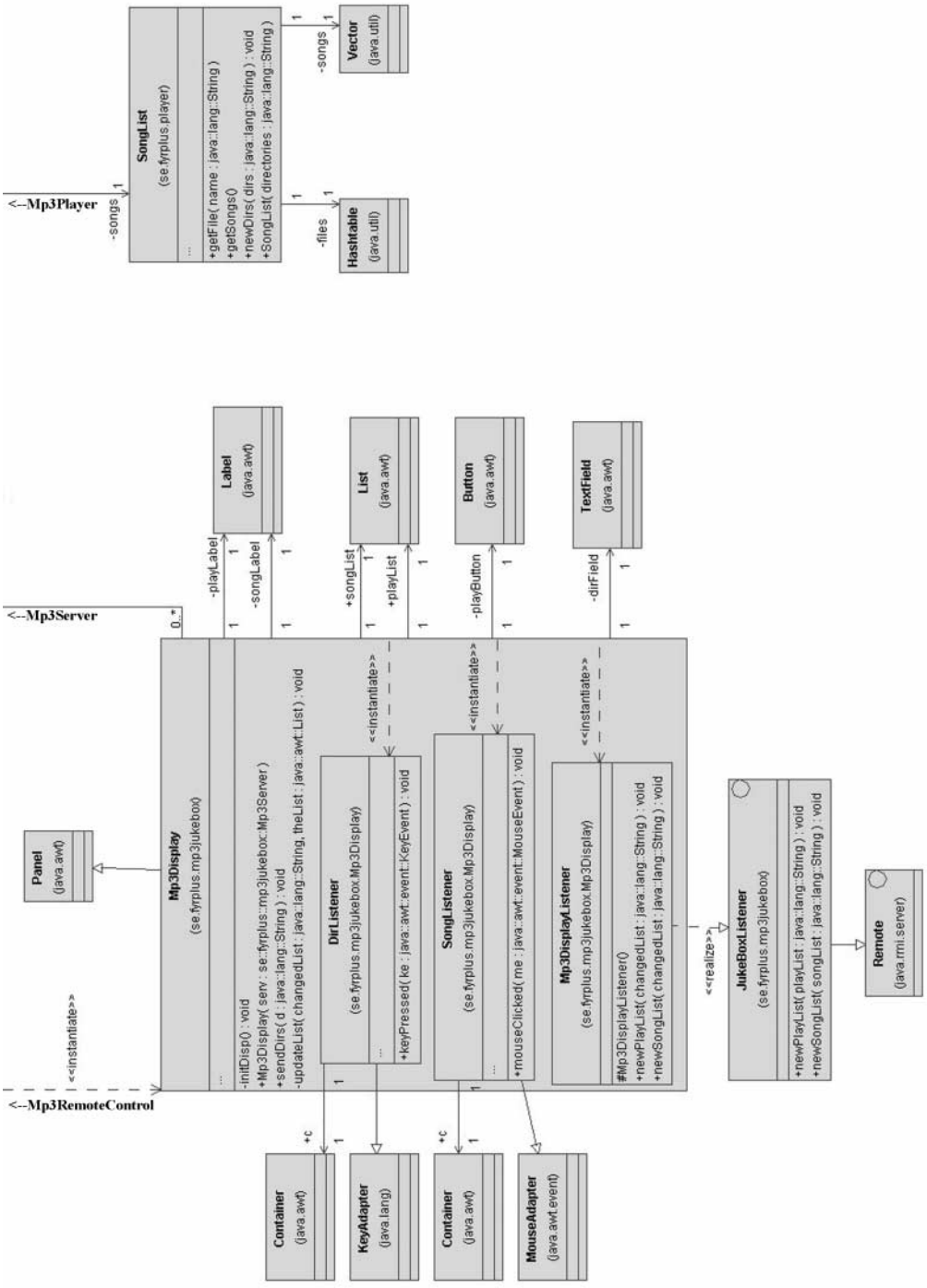

Figure 7.5.3: The server side of the application design, part 2.

Chapter 8

Application running

This chapter describes how the application is run. It also explains how to interpret the command lines needed to launch the server and the client. The last section shows a startup example from the development environment.

8.1 The client interface - the jukebox display

The client program is started by a command on the form:

```
java -Djava.rmi.server.codebase=http://{IP-address}:{portnumber}/
-Djava.security.policy={path to current policyfile}
{name of application}
```

The codebase[1] specifies where the class files are located on the client. The policy file states the rights of the program. The application name is *UniversalRemoteControl*. By executing this commandline the application starts and waits for the discovery process to find some suitable services and download these proxies. The client GUI is displayed in Figure 8.1.1.

[1] What codebase means and what it is used for can be read at http://java.sun.com/j2se/docs/guide/rmi/codebase.html.

The UniversalRemoteControl is able to show different kinds of proxy objects. These objects must be of type RemoteControl.

Figure 8.1.1: The client GUI.

8.2 The Mp3JukeBox server - the service

The server is run with the following parameters:

```
java -Djava.rmi.server.codebase=http://{IP-address}:{port number}/
-Djava.security.policy={path to policy file}
```

```
{name of application} -f {serviceidfilename} -d {1 to n mp3directories}
```

The codebase specifies where the class files are located on the server. The policy file states the rights of the application, this is important for security in the system. Here the name of the application is Mp3Jukebox. The rest of the commandline contains parameters to the program. The **-f** is for the serviceID file. The serviceid is a unique number assigned to the service when it is booted on the network. This file stores the number the first time the service is booted and is read the following times. If the file is nonexistent it is created. Otherwise it is read. Having the same serviceid every time the service is booted can be useful if the clients search for a specific service and use the attribute serviceID. The **-d** flag gives the application a list of directories to look for mp3-files in. The files found in these directories will make up the song list of the jukebox.

The network needs some kind of lookup service that is up and running. The lookup service keeps a registry of all the available services on the network. Sun's implementation of this service, Reggie, is sufficient for the purposes of this work and will be the lookup service of choice. Both the server and the client need a http-server that is responsible for loading the code.[2]

8.3 A startup example

A typical startup sequence using the existing tools is shown below.[3] Note that all directories are local to the development environment and that all output are expanded scripts.

1. Start the http server used to load classes for the lookup service:

   ```
   java -jar h:\jini\jini1_1\lib\tools.jar -port 8080
   -dir h:\jini\jini1_1\lib -verbose
   ```

[2] Classes that are Serialized and sent as bytecode.
[3] The startup order of the server and client does not matter.

2. Start the RMI daemon (rmid). The rmid is needed by the lookup service since this is an activatable[4] service.

```
rmid -J-Djava.security.policy=h:\jini\jini1_1\policy\policy.all
-log h:\log
```

3. Start the lookup service

```
set JINI_HOME=h:\jini\jini1_1
set HOSTNAME=172.16.1.77
set POLICYFILE=%JINI_HOME%\java.policy.all
set JARFILE=%JINI_HOME%\lib\reggie.jar
set CODEBASE=http://%HOSTNAME%:8080/reggie-dl.jar
set LOOKUP_POLICYFILE=%JINI_HOME%\example\lookup\policy\policy.all
set LOG_DIR=h:\temp\reggie_log
set GROUP=public
java -Djava.security.policy=%LOOKUP_POLICYFILE%
-jar %JARFILE% %CODEBASE% %LOOKUP_POLICYFILE% %LOG_DIR% %GROUP%
```

4. Start the httpserver pointing to the server program.

```
java -jar h:\jini\jini1_1\lib\tools.jar -port 8085
-dir h:\remote -verbose
```

5. Start the server program with all necessary arguments, note that the portnumber on the httpserver pointing to the server code must be the same as the portnumber for the codebase of the server.

[4] Activatable means that the service is only active when it is called. Rmid is responsible for the activation.

```
java -Djava.rmi.server.codebase=http://fyrpc177:8085/
-Djava.security.policy=h:\jini\jini1_1\java.policy.all
se.fyrplus.mp3jukebox.Mp3JukeBox -f h:\serviceidfile -d h:\mp3
```

6. Start the http server for the client.[5]

```
java -jar h:\jini\jini1_1\lib\tools.jar -port 8086 -dir h:\remote
-verbose
```

7. Start the client program

```
java -Djava.rmi.server.codebase=http://172.16.1.77:8086/
-Djava.security.policy=h:\jini\jini1_1\java.policy.all
se.fyrplus.universalremote.UniversalRemoteControl
```

[5]If the client and the server are colocated there is no need for two httpservers, they can share.

Chapter 9

Conclusion and summary

An application containing both Jini and Bluetooth technologies will have certain advantages in a communicating environment. First, the need for cables disappears. Second, explicit installations of device drivers is no longer needed. Third, one device can be used to control all other devices. These features will ease the communication and administration.

There are however some things that are worth noticing. Bluetooth is a relatively new technology and costs for education, development and manufacturing are still fairly high. The components are also still expensive.

Jini is free of charge and downloadable from Sun microsystems. It has an easy to understand application programming interface and it *does* make distributed computing easier. However, Jini does not work alone. In order to make Jini work properly a Java VM that supports at least some basic features (including RMI) need to be present. In addition a TCP/IP stack must be operational in the environment.

A product using the Jini-Bluetooth technology would be able to control all, for the user allowed, enabled devices in its close surroundings. There should no longer be a need to explicit load drivers into the device. This combination would enhance, for example, a mobile phone with the ablility of becoming a universal remote control containing only the drivers needed for the very moment. But, as seen above, Bluetooth components are

still too expensive and Jini, i.e. Java, demands a lot of resources of its environment. This mobile remote control would need to contain a Java VM with appropriate functionality, Jini classes, a TCP/IP stack and a Bluetooth stack. All these necessities demands a powerful, and probably expensive, device.

As a conclusion, the combination of these two technologies is not going to have any larger impact today, but when Bluetooth components, processor power and memory are cheap enough it probably will. At least there is potential.

The music jukebox application is an example of a distributed application. It contains a music server with an mp3 player and a remote control that controls the player by allowing the user of the control to choose songs that the player will play. The application uses the Jini and RMI technologies to transport data and discover services. The application is written in the Java language and is running over a TCP/IP network connection. The client and server class diagrams are modeled using the Unified Modeling Language (UML).

Running this kind of distributed application requires at least one web server, a lookup service, the rmi daemon and the application programs.

It is not a trivial task to set up the environment. And there is a lot of configuration to do before the application is functional. But when it does work, it works well.

Appendix A

Abbreviations

abbreviation	*meaning*
ACL	Asyncronous Connection-Less
AM_ADDR	Active Member Address
API	Application Programming Interface
AR_ADDR	Access Request Address
AWT	Abstract Window Toolkit
BD_ADDR	Bluetooth Device Address
BT	Bluetooth
CAC	Channel Access Code
Community	Set of services with connection
CRC	Cyclic Redundancy Check
CTS	Clear to Send
CVSD	Continuous Variable Slope Delta
DAC	Device Access Code
DH	Data High rate
DLCI	Data Link Connection Identifier
DM	Data Medium rate

DSR	Data Set Ready
DTR	Data Terminal Ready
DV	Data Voice
FEC	Forward Error Correction
FH	Frequency Hopping
FHS	Frequency Hopping Synchronization
FHSS	Frequncy Hopping Spread Spectrum
FIFO	First In First Out
GFSK	Gaussian Frequency Shift Keying
GUI	Graphical User Interface
HCI	Host Control Interface
HV	High quality Voice
HW	HardWare
IAC	Inquiry Access Code
I/O	Input/Output
IrDA	Infrared Data Association
JINI	Jini is not initials
JRE	Java Runtime Environment
JVM	Java Virtual Machine
L2CAP	Logical Link Control and Adaption Protocol
LAN	Local Area Network
LAP	Lower Address Portion
LC	Link Control channel
LM	Link Manager
LMP	Link Manager Protocol
LSB	Least Significant Bit
MAC	Medium Access Control sublayer

NAP	Non-significant Address Portion
PCM	Pulse Code Modulation
PDA	Personal Digital Assistant
PDU	Protocol Data Units
PM_ADDR	Parked Member Address
PPP	Point to Point Protocol
QoS	Quality of Service
RF	Radio Frequency
RMI	Remote Method Invocation
RS232	a serial communication interface
RTS	Request To Send
SCO	Syncronous Connection-Oriented
SDP	Service Discovery Protocol
SIG	(Bluetooth) Special Interest Group
TCS	Telephony Control Specification
TDD	Time Division Duplex
TCP	Transport Control Protocol
UA	User Asynchronous data channel
UAP	Upper Address Portion
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
UI	User Isosynchronous data channel
UML	Unified Modelling Language
US	User Synchronous data channel
USB	Universal Serial Bus
UUID	Universally Unique IDentifier
WAP	Wireless Application Protocol

Bibliography

[1] Peter Galvin Abraham Silberschatz. *Operating System Concepts, 5th Edition.* John Wiley and Sons, 1998.

[2] C.J Date. *An Introduction to Database Systems.* Addison-Wesley, Reading, Massachusetts, 1999.

[3] Digianswer A/S. Digianswer
www.digianswer.com.

[4] W. Keith Edwards. *Core Jini.* Prentice Hall, Upper Saddle River, NJ, 1999.

[5] Govind Seshadri. Java developer connection, the rmi tutorial.

[6] Bluetooth Special Interest Group. *The Profiles Specification.* Bluetooth SIG, Worldwide open specification, 1999.

[7] Bluetooth Special Interest Group. Baseband specification, core part b. Technical report, Bluetooth SIG, Feb 2001.

[8] Bluetooth Special Interest Group. Bluetooth protocol architecture, white paper. Technical report, Bluetooth SIG, Feb 2001.

[9] Bluetooth Special Interest Group. Blutooth audio, core appendix v. Technical report, Bluetooth SIG, Feb 2001.

[10] Bluetooth Special Interest Group. Host controller interface functional specification, core part h:1. Technical report, Bluetooth SIG, Feb 2001.

[11] Bluetooth Special Interest Group. IrDA interoperability, core part f:2. Technical report, Bluetooth SIG, Feb 2001.

[12] Bluetooth Special Interest Group. Link manager protocol , core part c. Technical report, Bluetooth SIG, Feb 2001.

[13] Bluetooth Special Interest Group. Logical link and adaption protocol specification, core part d. Technical report, Bluetooth SIG, Feb 2001.

[14] Bluetooth Special Interest Group. Radio specification, core part a. Technical report, Bluetooth SIG, Feb 2001.

[15] Bluetooth Special Interest Group. Rfcomm with ts 07.10 pecification, core part f:1. Technical report, Bluetooth SIG, Feb 2001.

[16] Bluetooth Special Interest Group. Service discovery protocol, core part e. Technical report, Bluetooth SIG, Feb 2001.

[17] James Rumbaugh Ivar Jacobson, Grady Booch. *The Unified Software Development Process*. Addison Wesley, One Jacob Way, Reading Massachusetts, 1999.

[18] Jamie Jaworski. *Java 2 Platform Unleashed*. Sams, 1999.

[19] Merriam Webster. Websters revised unabridged dictionary ftp://ftp.uga.edu/pub/misc/webster/.

[20] Bob O'Hara and Al Petrick. *IEEE 802.11 Handbook,A designers companion*. Standards Information Network, IEEE press, http://standards.ieee.org, 1999.

[21] W. Richard Stevens. *TCP/IP Illustrated, Volym 1, The protocols*. Addison-Wesley publishibg Company, One Jacob Way, Massachusetts, 1994.

[22] Sun Microsystems. Rmi specification. http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmitoc.html.

[23] Andrew S. Tanenbaum. *Computer Networking,third edition*. Prentice Hall, Inc, Upper Saddle River, New Jersey, 1996.

[24] The Jini Community. What is jini network technology. http://www.jini.org/whatisjini.html.

[25] The Salutation Consortium. Salutation www.salutation.org.

[26] Ken Arnold Bryan O'Sullivan Robert W. Scheifler Jim Waldo Ann Wollrath. *The Jini Specification*. Addison-Wesley, Reading, Massachusetts, 1999.

[27] Scott Oaks Henry Wong. *Jini in a nutshell*. O'Reilly, Sebastopol, CA, 2000.